Control structures, second lecture

Non-local control:
from subroutines to functions and coroutines

Xavier Leroy

2024-02-01

Collège de France, chair of software sciences
`xavier.leroy@college-de-france.fr`

Subroutines, procedures, functions

Sharing and reusing code

Some computations occur repeatedly!

```
D = SQRT(B*B - 4*A*C)
X1 = (-B + D) / (2*A)
X2 = (-B - D) / (2*A)
```

How can we write this code once and just “call it” whenever we need to run it?

Subroutines in assembly language

Easy to write using a computed jump instruction.

```
; Solve quadratic equation AX^2 + BX + C = 0
; Input: A in r1, B in r2, C in r3, return address in r4
; Output: solutions in r1 and r2
quadratic:
        mul r5, r2, r2      ; compute solutions
        ...
        jump r4             ; return to caller
```

Call sites:

```
        mov r4, L100        ; set return address
        branch quadratic    ; invoke subroutine
L100:   ...                 ; execution resumes here
```

Subroutines in assembly language

Most processors provide a `call` instruction that jumps to a given code address while saving the address of the next instruction in a register or on a stack.

```
call quadratic, r4    ; first invocation
...
call quadratic, r4    ; second invocation
...
```

To handle nested calls, use different registers or save the return addresses in memory, e.g. on a call stack.

Subroutines in FORTRAN I

Like in assembly language, using the computed `goto` statement (`ASSIGN` *label* `TO` *var* … `GO TO` *var*)

```
200:  D = SQRT(B*B - 4*A*C)
      X1 = (-B + D) / (2*A)
      X2 = (-B - D) / (2*A)
      GO TO RETADDR

1000: A = ... B = ... C = ...
      ASSIGN 1010 TO RETADDR
      GO TO 200
1010: PRINT X1
```

Subroutines and functions in Fortran II

Fortran II (1958) introduces language support for defining subprograms with explicit parameters.

1- Subroutines:

```
SUBROUTINE QUADRATIC(A, B, C, X1, X2)
   D = SQRT(B*B - 4*A*C)
   X1 = (-B + D) / (2*A)
   X2 = (-B - D) / (2*A)
   RETURN
END
```

Invocation: `CALL QUADRATIC(1.0, -2.0, 5.0, X1, X2)`

Arguments that are variables or arrays are passed by reference.

All variables are local to a sub-program or to the main program, unless declared `COMMON`.

Subroutines and functions in Fortran II

2- Simple functions: an expression with parameters.

```
INTPOL(X) = A * X + B * (1 - X)
X2 = INTPOL(0.5)
X3 = INTPOL(0.333333)
```

3- General functions: a subroutine + a return value.

```
FUNCTION AVRG(ARR, N)
DIMENSION ARR(N)
   SUM = ARR(1)
   DO 10 I=2, N
   SUM = SUM + ARR(I)
10: AVRG = SUM / FLOATF(N)
   RETURN
END
```

Invocation: `X = AVRG(A,20) + AVRG(B,10)`

Procedures and functions in Algol 60

Close to subprograms in Fortran II:

- a procedure = a command with parameters;
- a function = a procedure with a return value.

Main differences:

- arguments are passed by value or by name;
- procedures can be nested and can access the variables of the enclosing procedure;
- recursion is explicitly supported;
- a procedure can be passed as an argument to another procedure.

A procedure in Algol 60

```
procedure quadratic(a, b, c, x1, x2);
     value a, b, c;
     real a, b, c, x1, x2;
begin
  real d;
  d := sqrt(b * b - 4 * a * c);
  x1 := (-b + d) / (2 * a);
  x2 := (-b - d) / (2 * a)
end;
```

Nested functions, functions as arguments

```
real procedure test(a, b);
    value a, b; real a, b;
    begin
        real procedure interpolate(x);
        value x; real x;
        begin
            interpolate := a * x + b * (1 - x)
        end;
        test := integrate(interpolate, 0.0, 10.0)
    end
```

Call by name

The famous copy rule:

> *Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter … Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved … Finally the procedure body, modified as above, is inserted in place of the procedure statement [the call] and executed …*
>
> *(Report on the Algorithmic Language ALGOL 60)*

Close to call-by-name in the lambda-calculus, and to hygienic macros in Scheme. Exhibits some surprising behaviors!

Greatness of the copy rule

A versatile summation function:

```
real procedure Sum(k, l, u, ak)
     value l, u; integer k, l, u; real ak;
  begin
     real s;
     s := 0;
     for k := l step 1 until u do
        s := s + ak;
     Sum := s
  end;
```

Sum of array `A`: `Sum(i, 1, m, A[i])`

Sum of squares: `Sum(i, 1, n, i*i)`

Sum of matrix `A`: `Sum(i, 1, m, Sum(j, 1, n, A[i,j]))`

Misery of the copy rule

```
procedure swap(a, b)
    integer a, b;
    begin
        integer temp;
        temp := a;
        a := b;
        b := temp;
    end;
```

This procedure does not always exchange its arguments!

For instance, `swap(i, A[i])` expands to

`temp := i; i := A[i]; A[i] := temp`.

(→ A move towards call-by-value + call-by-reference in post-Algol languages such as Pascal, Ada, C++, …)

Design dimensions for procedures and functions

Some possible choices:

- Semantics of argument passing
  (by value, by reference, by pointer, by name, ...)
- Recursion and reentrancy (or not)
- Nested functions (or not)
- Scoping of variables (lexical, dynamic)
- Lifetimes of variables (one block, the whole program, ...)
- Functions as values
  (first-class, or only as arguments to other functions).

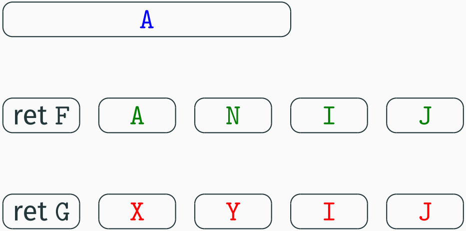
The choices are tied to the implementation techniques for the environments that maintain the values of variables.

Statically-allocated environments (FORTRAN)

```
DIMENSION A(10)
COMMON A

SUBROUTINE F(A, N)
  ... I ... J...

SUBROUTINE G(X, Y)
  ... I ... J ...
```



A

ret F A N I J

ret G X Y I J

One memory location per `COMMON` variable.

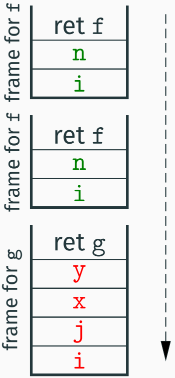
One memory location per variable of a subroutine.

One memory location per subroutine to hold the return address.

Simple and efficient, but does not support recursion.

Using a stack of activation records (stack frames)

```
procedure g(x, y)
begin
  integer i, j;
  ...
end;
procedure f(n)
begin
  integer i;
  ... f( ) ... g( ) ...
end
```

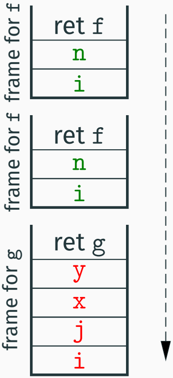


The stack frame for a function activation contains its local variables (unless declared `static`) and its return address.

Function call = push a frame; function return = pop this frame.

Using a stack of activation records (stack frames)

```
procedure g(x, y)
begin
  integer i, j;
  ...
end;
procedure f(n)
begin
  integer i;
  ... f( ) ... g( ) ...
end
```



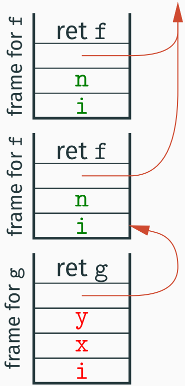
Without nested functions (as in C):

environment = current stack frame for the function
+ global and `static` variables;

function value = pointer to its code.

Stack frames for nested functions

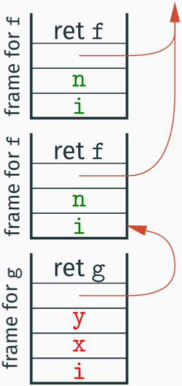
```
procedure f(n)
begin
  integer i;
  procedure g(x, y)
  begin
    integer j;
    ...
  end;
  ... f( ) ... g( ) ...
end
```



Chaining of the most recent stack frames for the enclosing functions. When a function is called, the head of the chain is passed as an extra argument.

Stack frames for nested functions

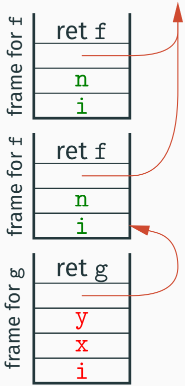
```
procedure f(n)
begin
  integer i;
  procedure g(x, y)
  begin
    integer j;
    ...
  end;
  ... f( ) ... g( ) ...
end
```



Environment = current stack frame for the function
+ current stack frames for enclosing functions
+ global or `static` variables

Stack frames for nested functions

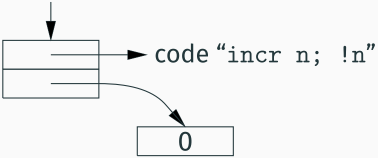
```
procedure f(n)
begin
  integer i;
  procedure g(x, y)
  begin
    integer j;
    ...
  end;
  ... f( ) ... g( ) ...
end
```



Function value = code pointer + head of stack frame chain
(≈ a closure of the code by the environment).

Heap allocation of function closures and objects

```
let counter () =
  let n = ref 0 in
  fun () -> incr n; !n
```



Supports using as first-class values
function closures (functions with free variables) or
objects (set of methods sharing some instance variables).

Decouples the lifetimes of variables from the call stack discipline.

Control flow around function calls

Control flow for a procedure/function call

In Fortran II as in many later languages, the flow of control around a procedure call is simple:

- when the procedure returns, execution continues with the command that follows (syntactically) the call;
- labels are local to procedures
  → no `goto` jumps from a procedure to another.

In other words, the invocation `CALL` $proc(e_1, \dots, e_n)$
is a base command, like an assignment $x := e$
(except that the call may not terminate).

Procedure with multiple return points

In Fortran 77, a procedure can have other return points besides the point following the `CALL`. These alternate return points are labels passed as extra arguments.

```
SUBROUTINE QUADRATIC(A, B, C, X1, X2, *)
    D = B*B - 4*A*C
    IF (D .LT. 0) RETURN 1
    D = SQRT(D)
    X1 = (-B + D) / (2*A)
    X2 = (-B - D) / (2*A)
    RETURN
END
    ...
    CALL QUADRATIC(1.0, -2.0, 12.5, X1, X2, *99)
    ...
99: WRITE (*,*) 'Error - no real solutions'
    STOP
```

Non-local “goto”

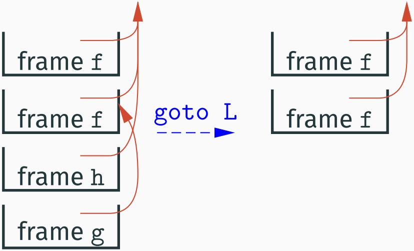
In Algol and Pascal, a `goto L` can exit one or several enclosing blocks, as long as the `goto` is in the scope of the definition of `L`.

```
begin
    ...
    begin
        integer i;
        ... goto L ...
    end;
 L: ...
end
```

This works even if `goto L` is in a procedure defined in the scope of `L`.

Exiting a procedure with a “goto”

```
procedure h(p)
begin
L: ... p() ...
end;
procedure f(n)
begin
  procedure g()
  begin goto L end;
  ... f() ... h(g) ...
L:...
end
```



The non-local `goto L` terminates procedure `g` and the previous procedure activations, until it comes back to the activation that defines `L`, i.e. the latest activation of `f`.

Example: fatal errors in the Pascal source of TEX

```
label end_of_TEX, final_end;

procedure jump_out;
begin goto end_of_TEX;
end;

begin
  ...
end_of_TEX: close_files_and_terminate;
final_end: ready_already:=0;
end.
```

Multiple return points and non-local "goto"

In Pascal, we cannot pass a label `L` as a parameter, but we can pass a procedure that performs `goto L`.

```
procedure quadratic(a, b, c: real; var x1, x2: real;
                    esc: procedure ());
variable d: real;
begin
   d := b * b - 4 * a * c;
   if d < 0 then esc();
   d := sqrt(d);
   x1 = (-b + d) / (2*a);
   x2 = (-b - d) / (2*a)
end;
```

Multiple return points and non-local “goto”

```
procedure solve(a, b, c: real);
variable x1, x2: real;
label error, done;

    procedure goto_error;
    begin goto error end;

begin
    quadratic(a, b, c, x1, x2, goto_error);
    writeln('Solutions:', x1, x2);
    goto done;
  error:
    writeln('No real solutions');
  done:
end;
```

Extra result vs. multiple return points

A more popular approach: return an extra result (result code, error code) indicating how the function terminated (normally or on an error).

```
int quadratic(double a, double b, double c,
              double * x1, double * x2)
{
    double d = b * b - 4 * a * c;
    if (d < 0) return -1;
    d = sqrt(d);
    *x1 = (-b + d) / (2 * a);
    *x2 = (-b - d) / (2 * a);
    return 0;
}
```

Using return codes

```
void solve(double a, double b, double c)
{
    double x1, x2;
    int rc = quadratic(a, b, c, &x1, &x2);
    if (rc < 0) {
      printf("Error - no real solutions\n");
      exit(2);
    }
    printf("Solutions: %f %f\n", x1, x2);
}
```

✔ Handling the error at point of call.

Using return codes

```
int solve(double a, double b, double c)
{
    double x1, x2;
    int rc = quadratic(a, b, c, &x1, &x2);
    if (rc < 0) {
      return -1;

    }
    printf("Solutions: %f %f\n", x1, x2); return 0;
}
```

- ✓ Handling the error at point of call.
- ✓ Propagating the error code towards the caller.

Using return codes

```
void solve(double a, double b, double c)
{
    double x1, x2;
    int rc = quadratic(a, b, c, &x1, &x2);




    printf("Solutions: %f %f\n", x1, x2);
}
```

- ✔ Handling the error at point of call.
- ✔ Propagating the error code towards the caller.
- ✘ Ignoring the error and proceeding as if nothing happened.

The “option” and “result” types

Prevent programmers from ignoring errors by using sum types and strong typing.

A common idiom in functional languages and in Rust.
E.g. in OCaml:

```
type ’a option = Some of ’a | None

type (’a, ’e) result = Ok of ’a | Error of ’e

let quadratic a b c : (float * float) option =
    let d = b *. b -. 4. *. a *. c in
    if d < 0.0 then None else
        let d = sqrt d in
        Some((-. b +. d) /. (2. *. a), (-. b -. d) /. (2. *. a))
```

The “option” and “result” types

Static typing and exhaustiveness of pattern matching make it impossible to ignore errors:

```
let solve a b c =
   match quadratic a b c with
   | Some(x1, x2) ->
       printf "Solutions: %f %f\n" x1 x2
   | None ->
       printf "Error - no real solutions\n"
```

Propagating the error towards the caller is achieved by clauses `| None -> None` or `| Err reason -> Err reason’`

Haskell, OCaml, Rust provide lightweight syntax for this (monadic notations, Rust’s “?” operator, etc).

Structured exceptions and exception handlers

An exception = a data structure describing an exceptional condition (error, absence of a result value, …).

Two language constructs:

- Raising / throwing an exception: `throw` *exn*

  abort the current computation and send the exception to the first enclosing handler.

- Handling / catching exceptions: `try` s_1 `catch(...)` s_2

  intercept exceptions raised during the execution of command s_1 and executes command s_2.

Example of structured exception handling in Java

```
static double[] quadratic(double a, double b, double c)
throws NoSolution
{
   ... throws (new NoSolution()); ...
}
static void solve(double a, double b, double c)
{
    try {
      double[] sols = quadratic(a, b, c);
      System.out.println(
          "Solutions: " ++ sols[0] ++ ", " ++ sols[1]);
    } catch (NoSolution e) {
      System.out.println("No real solutions");
    } finally {
      System.out.println("I'm done!");
    }
}
```

Intuitive semantics for structured exceptions

`throw` within the body of a `try`:

≈ `break` for early termination of a block (*multi-level exit*);
≈ forward `goto`.

`throw` in a function without a `try`:

- dynamic search of the call stack for a caller with a `try` that can handle the exception;
- execution of the `finally` clauses of the `try` that were skipped.

Compared with a non-local `goto`: the handler is determined dynamically, instead of being determined by the code that raises the exception.

A brief history of structured exception handling

1972 MacLisp: `THROW`, `CATCH`, then `UNWIND-PROTECT` ($\approx$ `try`…`finally`).

1975 J. B. Goodenough. *Exception handling: issues and a proposed notation*, CACM 18(12).

1975 CLU (B. Liskov, MIT).
(Declaration of exceptions that can escape a function, with dynamic checking.)

1978 LCF ML and its descendants (SML, Caml, …).
(No declarations.)

1980 Ada
(No declarations.)

1990 C++
(Optional declarations, obsoleted in C11, removed in C17.)

1995 Java
(Mandatory declarations, with static checking)

The controversy around exceptions

Pros:

- No need to write code to obtain the most common behavior, i.e. the propagation of exceptions towards the caller.
- Clearly separates the code that detects an error from the code that is able to handle the error.

Cons:

- Creates control flows that are not visible in the source code.
- Too easy to forget to handle exceptions.
- Difficult to finalize resources in presence of exceptions.

(See Stroustrup's note given in reference, and lecture #7.)

Inverting or symmetrizing control: iterators, generators, coroutines

Example: print a linked list of integers

In C:

```
for (list l = lst; l != NULL; l = l->next)
    printf("%d\n", l->val);
```

In OCaml:

```
List.iter (fun n -> printf "%d\n" n) lst
```

In Java:

```
for (Iterator<Int> i = lst.iterator(); i.hasNext(); ) {
   System.out.println(i.next())
}
```

In Python:

```
for n in lst: print(n)
```

Two ways to abstract over the traversal of a data structure

“Internal” iterator:

a higher-order function that calls the user-provided code.

```
List.iter: ('a -> unit) -> 'a list -> unit
List.map: ('a -> 'b) -> 'a list -> 'b list
List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

“External” iterator:

user code calls the methods of an “iterator” object.

```
interface Iterator<T> {
    boolean hasNext();
    T next();
}
```

This is called control inversion: *don’t call us, we’ll call you!*

The flexibility provided by external iterators

Make it easy to traverse several data structures at the same time.

Example: the same fringe problem (determine whether two binary search trees contain the same values).

```
boolean same_fringe(TreeSet<T> s1, TreeSet<T> s2) {
    Iterator<T> i1 = s1.iterator();
    Iterator<T> i2 = s2.iterator();
    while (i1.hasNext() && i2.hasNext()) {
        if (! i1.next().equals(i2.next())) return false;
    }
    return ! i1.hasNext() && ! i2.hasNext();
}
```

Implementing an external iterator

Easy in an object-oriented language: use instance variables of the iterator object to “remember where we are” in the traversal.

```
class ArrayIterator<T> {
    private T[] arr;
    private int i;
    boolean hasNext() { return i < arr.length; }
    T next() { T res = arr[i]; i++; return res; }
    ArrayIterator(T [] arr) { this.arr = arr; this.i = 0; }
}
```

(In red: the parts of the code that would also occur in a direct traversal with a `for` loop.)

Implementing an external iterator

Easy as well in a functional/imperative language: use functions with free mutable variables as first-class values.

```
let array_iterator (arr: 'a array) : unit -> 'a option =
    let i = ref 0 in
    fun () ->
        if !i >= Array.length arr
        then None
        else (let res = arr.(!i) in incr i; Some res)
```

Generators (in the style of Python)

A way to write iterators in direct style, as functions that return successive results at each call.

```
def array_elements(a):
    i = 0
    while i < len(a):
        yield a[i]
        i += 1
```

`yield` *v* : return value *v* to the caller; the function execution can restart later just after the `yield`.

`return` *v* : return value *v* to the caller; terminates the function execution.

Generators (in the style of Python)

A way to write iterators in direct style, as functions that return successive results at each call.

```
def array_elements(a):
    i = 0
    while i < len(a):
        yield a[i]
        i += 1
```

Examples of use:

```
for i in array_elements((1,2,3)): print(i)

g = array_elements((1,2,3))
print(next(g))
print(next(g))
```

Producing an infinite sequence on demand

```
def primes():
    """Generator for prime numbers"""
    p = [2]; yield 2
    m = 3
    while True:
        i = 0
        while i < len(p) and p[i] * p[i] <= m:
            if m % p[i] == 0: break
            i += 1
        else:
            p.append(m); yield m
        m += 2
```

Generators for non-determinism and error reporting

Non-determinism $\approx$ several return values are possible.

Error $\approx$ lack of a return value.

```
def quadratic(a, b, c):
    """Generate the solutions of ax^2 + bx + c = 0"""
    d = b * b - 4 * a * c
    if d < 0: return
    d = math.sqrt(d)
    yield ((-b - d) / (2 * a))
    if d != 0: yield ((-b + d) / (2 * a))
```

Compiling a generator

Idea: a remanent variable of “code pointer” type, where we store the code address (the label) that follows the `yield`.

```
def generator():
    n = 0; while True: yield n; yield (-n); n += 1
```

In GNU C (where labels can be used as values):

```
int generator(void) {
      static void * pc = &&start;
      static int n;
      goto *pc;
  start: n = 0; while (true) {
          pc = &&yield1; return n; yield1:
          pc = &&yield2; return (-n); yield2:
          n += 1;
      }
}
```

Stackless generators vs. stackful generators

Example: enumerate the values at the nodes of a binary tree, following an infix traversal.

```
def inorder(t):
    if t:
        inorder(t.left)
        yield t.val
        inorder(t.right)
```

Stackless generators vs. stackful generators

Example: enumerate the values at the nodes of a binary tree, following an infix traversal.

```
def inorder(t):
    if t:
        inorder(t.left)
        yield t.val
        inorder(t.right)
```

Doesn't work, because Python's generators are stackless. Recursive calls to `inorder` create new generators, which are unused. A single value is returned, that of the top of the tree.

Alternatives: pipelining generators (Python's `yield from`), or a different syntax and a different implementation for stackful generators, with a call stack that persists between `yield`.

Asymmetric coroutines vs. symmetric coroutines

Asymmetric coroutines: another name for stackful generators.

- distinguish callee (generator) from caller (consumer);
- `yield` branches back to the caller.

Symmetric coroutines: a kind of cooperative threads.

- all coroutines stand “at the same level”;
- `yield` passes control to an explicitly-specified coroutine.

(Simula, Modula-2)

An example of symmetric coroutines

```
q = queue.Queue(maxsize = 100)

coroutine produce():
    while True:
        while not q.full(): item = build(); q.put(item)
        yield to consume

coroutine consume():
    while True:
        while not q.empty(): item = q.get(); use(item)
        yield to produce

produce()
```

The same example with cooperative threads

```
def produce():
    while True:
        while q.full(): yield
        item = build(); q.put(item)
        yield

def consume():
    while True:
        while q.empty(): yield
        item = q.get(); use(item)
        yield

spawn(produce); spawn(consume)
```

The interleaving of computations is partially left to the scheduler.

An analysis of coroutines by de Moura and Ierusalimschy

(Ana Lúcia de Moura and Roberto Ierusalimschy, *Revisiting Coroutines*, TOPLAS 31(2), 2009.)

Three design dimensions:

- asymmetric / symmetric coroutines; (semantics of `yield`)
- stackful / stackless coroutines; (position of `yield`)
- as first-class values or limited to e.g. `for` loops.

An analysis of coroutines by de Moura and Ierusalimschy

(Ana Lúcia de Moura and Roberto Ierusalimschy, *Revisiting Coroutines*, TOPLAS 31(2), 2009.)

Three design dimensions:

- asymmetric / symmetric coroutines; (semantics of `yield`)
- stackful / stackless coroutines; (position of `yield`)
- as first-class values or limited to e.g. `for` loops.

Main result:

Asymmetric, stackful, first-class coroutines
have the expressive power of one-shot delimited continuations
and can encode all the other control structures seen today.

(→ Lectures #4 and #5)

Examples of encodings

Symmetric coroutines encoded with asymmetric coroutines:
`yield to` *C* becomes `yield` of value *C* to a trampoline.

```
c = first generator
while True: c = next(c)
```

Cooperative threads encoded with asymmetric coroutines:
a scheduler calls the coroutines in round-robin manner.

```
while not q.empty():
    c = q.get()
    try: next(c); q.put(c)
    except StopIteration: pass
```

Summary

Summary

Subroutines, procedures, functions and methods remain even today the main language construct to support the decomposition of programs in pieces that are reusable and understandable independently.

The corresponding control flow (call – compute – return) is simple… except when it is not:

- multiple returns, non-local jumps, …;
- structured exceptions and exception handlers;
- control inversion: iterators, generators;
- control symmetrization: symmetric coroutines, threads.

References

References

An analysis and a formalization of coroutines:

- Ana Lúcia de Moura and Roberto Ierusalimschy, *Revisiting Coroutines*, TOPLAS 31(2), 2009.

A discussion of exceptions vs. return codes:

- Bjarne Stroustrup, *C++ exceptions and alternatives*, note P1947, 2019.