Subprograms

Akim Demaille akim@lrde.epita.fr
Roland Levillain roland@lrde.epita.fr

EPITA — École Pour l'Informatique et les Techniques Avancées

June 14, 2012

Subprograms

Subprograms

- At the origin, snippets copied and pasted from other sources [4, Chap. 5].
  - 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.

Vocabulary

Procedure Subprograms with no return value.

Function Subprograms that return something.

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

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)
```

Please reserve "parameter" for templates.

Routines

Swap in Fortran

```
SUBROUTINE SWAP (I1, I2)
INTEGER I1, I2, TMP
TMP = I1
I1 = I2
I2 = TMP
RETURN
END
```

```
PROGRAM SWAPPING
INTEGER BIG, SMALL
INTEGER HEAD, TAIL
...
CALL SWAP (BIG, SMALL)
...
CALL SWAP (HEAD, TAIL)
...
STOP
END
```

No type checking between formal and effective arguments.

Swap in C

```
void
swap (int *ip1, int *ip2)
{
  int tmp = *ip1;
  *ip1 = *ip2;
  *ip2 = tmp;
}
```

```
int
main (void)
{
  int big, small, head, tail;
  ...
  swap (&big, &small);
  ...
  swap (&head, &tail);
  ...
  return 0;
}
```

Type matching checked since ISO-C.

Swap in Modula 2

```
MODULE swapping
  VAR big, small, head, tail: INTEGER;

PROCEDURE SWAP (VAR i1, i2: INTEGER);
  VAR tmp: INTEGER;
  BEGIN
    tmp := i1;
    i1  := i2;
    i2  := tmp;
  END swap
```

```
BEGIN
  ...
  swap (big, small);
  ...
  swap (head, tail);
  ...
END swapping.
```

Embedded declaration of swap in swapping.

Swap in Ada

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

```
begin
  ...
  swap (big, small);
  ...
  swap (head, tail);
  ...
end swapping;
```

Local declaration.

Communication in Fortran

```
PROGRAM MAIN        SUBROUTINE A ...    SUBROUTINE B ...
COMMON X, Y, Z      COMMON U, V, W      COMMON E, F, G
INTEGER R           INTEGER S           INTEGER T
...                 ...                 ...
STOP                RETURN              RETURN
END                 END                 END
```

- The `COMMON` part is shared, independently of the names types, and length of the `COMMON` declaration...
- Memory reduction (*overlaying*).
- Danger.
- Independent compilation ($\neq$ separated).

Communication in Algoloids

```
program main ...
var r, w: real;
  procedure a...
    var x, y: real;
    procedure b ...
      var x, y: real;
      begin ... end;
    begin ... end;

   procedure c ...
   var w, x: integer;
   begin ... end;

begin ... end.
```

- Block structure (Algol 60, Pascal, Modula 2, Ada, Tiger).
- Static scoping (contrary to Sh, Perl etc.).
- Declaration in blocks ($\neq$ block structure): most languages (Algol 60, C, etc.)
- Non separated compilation, but typing.
- Closed scope: scope control (e.g., Euclid). Identifiers must be imported.

Argument passing

Argument passing

From a naive point of view, 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		*	*		*		

Argument passing from a subprogram

By Result `out` in Ada, sort of a non initialized local variable. *On return*, the effective argument (an l-value) is updated. Assignments only:

```
procedure negative_get (negative : out integer) is
  number : integer
begin
  get (number);
  while number >= 0 do
    put_line ("Try again!");
    get (number);
  end loop;
  negative := number;
end negative_get;
```

Algol W: lvalue evaluated on return, in Ada, on call.
Problem: local copy. The compiler optimizes.

Passing from/to a subprogram

By Value-Result `in out` `in` Ada, exact combination of `in` and `out`: local copy.

By Reference Work directly onto the effective argument, via an indirection.

- By Value-Result: copies.
- By Reference: indirections.
- Usually indistinguishable, except with synonymy (*aliasing*), and concurrent programming.
- Pascal forbids `swap (foo, foo)`, but what about `swap (foo[bar], foo[baz])`...
- In Fortran, the lvalue may be a...simple rvalue...

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.

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

Some sugar

In Ada, named arguments and/or default values:

```
-- Output a float with a specified precision.
put (number : in float,
     before : in integer := 2, after : in integer := 2,
     exponent : in integer := 2)

...
begin
  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);
end
```

In Perl, use a hash as argument.

Functions

Functions

Fortran

```
INTEGER FUNCTION SUM(A, B)
INTEGER A, B
SUM = A + B
RETURN
END
```

Algol 60

```
integer procedure sum(a, b);
value a, b; integer a, b;
sum := a + b;
```

Pascal

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

Functions

Modula 2

```
PROCEDURE sum(a, b: INTEGER): INTEGER;
BEGIN
  RETURN a + b;
END sum
```

Ada

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

Functions: Side effects

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

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

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

Functions: Side effects

Possible in Eiffel but strongly against its culture.

```
feature -- To read one character at a time:

   read_character is
         -- Read a character and assign it to ‘last_character’.
      require
         is_connected
         not end_of_input
      deferred
      ensure
         not push_back_flag
      end

   last_character: CHARACTER is
         -- Last character read with ‘read_character’.
      require
         is_connected
      deferred
      end
```

Functions: Side effects

In Ada

- (`in`) `out` are forbidden in functions.
- Globals are still there. . .

Functions as Values

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 [2]...

```
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,
   'Werror'            => sub { $SIG{"__WARN__"} = sub { die $_[0] } },
   'Wno-error'         => sub { $SIG{"__WARN__"} = 'DEFAULT' },
  )
    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 res end
    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 [1, Chap. 15].

Strictness

Call by name [1, Chap. 15]

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
  f (loop (y))  /* ≡ 'if y > 8 then loop (y) else -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 [3]

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

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

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)
```

Lazy evaluation 2 [3]

```
elimerror n (cons a (cons b rest)) =
  = cons ((b * (2 ** n) - a) / (2 ** n - 1))
         (elimerror n (cons b rest))
```

What is the value of n?

```
order (cons a (cons b rest)) =
  = round (log2 ((a - c) / (b - c) - 1))
```

thus

```
improve s = elimerror (order s) s
within eps (improve (differentiate h0 f x))
```

and actually

```
super s = map second (repeat improve s)
second (cons a (cons b rest)) = b

within eps (super (differentiate h0 f x))
```

Bibliography I


Andrew W. Appel.
Modern Compiler Implementation in C, Java, ML.
Cambridge University Press, 1998.


Alexandre Duret-Lutz and Tom Tromey.
GNU Automake.
`http://www.gnu.org/software/automake/`, 2003.


John Hughes.
Why functional programming matters.
The Computer Journal, 32(2):98–107, April 1989.
`http://www.math.chalmers.se/~rjmh/Papers/whyfp.html`.

Bibliography II


Leslie B. Wilson and Robert G. Clark.
Langages de Programmation Comparés.
Addison-Wesley, 2nd edition, November 1993.