Subprograms

Contents

General concepts

Originally, subprograms (procedures and functions) were a mechanism for code reuse—a way to avoid repetitions. Now they are seen as a fundamental abstraction mechanism.

A subprogram is a named block (with a local scope) that encapsulates an algorithm.

Semantically, subprogram is a complex operation that can be initiated (called) as if it were an elementary operation. This is referred to as process abstraction. A subprogram:

- is defined by means of lower-level operations,
- has a name,
- has a method of accepting arguments / delivering results (parameter passing and communication through non-local objects).
- is computed by suspending the calling program unit, executing the block's statement and returning to the caller.

Subprogram header (heading): name, parameter-passing modes, the type of the value (if the subprogram is a function).

Subprogram body: a sequence of statements.

A subprogram may be defined (declared), but never called. On the other hand, it may be called many times, also recursively.

A procedure extends the language: it is a new kind of statement. Similarly, a function is a new kind of operation.

A parameter of a subprogram is a generalization of an object manipulated by the statements—a subprogram is supposed to work in the same way for every parameter value. We also talk about "dummy variables".

By parameter passing we mean replacing generalized entities in a subprogram declaration (formal parameters) with existing objects (actual parameters).

Parameter passing

A parameter passing mode determines how much of the argument is given to the subprogram:

only value — only address — address and value

and what are the restrictions on the use of the argument:

read-only — write-only — read/write.

Pass-by-Value

(`in` parameters in Ada; value parameters in Pascal; all parameters in Algol-68, C)

Only the value of the parameter is given to the subprogram. To store this value, we use a local variable—its address.

This is usually implemented by computing and copying the actual parameter's value into the subprogram's memory space. This may be expensive if the parameter is a large object, for example, an array.

Pass-by-Result

(`out` parameters in Ada; not available in Pascal)

The address (but not the value) of the parameter is made available to the subprogram—for write-only access. Technically, this is a local object that cannot appear in expressions.

Pass-by-Value-Result

(`in-out` parameters in Ada; not in Pascal)

(Also called Pass-by-Copy). When the subprogram is activated, the value of the parameter is copied into a local object. The final value of this object is copied back into the parameter.

```
-- P(in out X: char), the call is P(Z):
begin  X := Z;
       -- body of P
       Z := X;
end;
```

This method and Pass-by-Result may—in a particularly bizarre situation—be sensitive to the order in which parameters are evaluated (see the textbook).

Pass-by-Reference

(`var` parameters in Pascal; not available in Ada; all parameters in older Fortran)

The address and value are both available to the subprogram's statements.

This is implemented by indirect addressing: the actual parameter is the address of a variable in the calling program unit. No copying is necessary!

A big potential problem with Pass-by-Reference: aliasing. One possible situation: passing the same variable as two different actual parameters. Another possibility: a non-local variable referenced in the subprogram and passed as a parameter.

[How does Pass-by-Reference mode differ from Pass-by-Value-Result?]

Note:

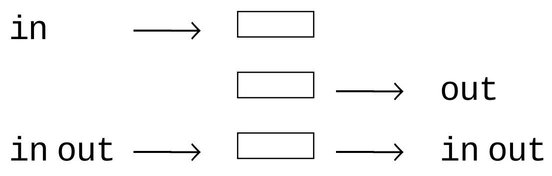
A parameter passed by Result, Value-Result, Reference must be a variable.

Example: MODE can be **in**, **out**, **in out**, **var**

```
program P;
var J: integer;
    A: array [1..10] of integer;
procedure SWAP(MODE X,Y: integer);
var TMP: integer;
begin
  TMP := X;  X := Y;  Y := TMP;
end;
begin
  J := 3;  A[J] := 6;
  SWAP(J, A[J]);
  writeln(J, A[3]);
end.
```

MODE = `in`:
`J` = `3`, `A[3]` = `6` (no change, as expected)
MODE = `out`:
impossible (we can't read the value of `X`)
MODE = `var`, `in out`:
`J` = `6`, `A[3]` = `3` (a correct swap)

A simplified model of a procedure:
a “black box” that reads in values and writes out results.



The classification of formal parameters based on addresses and values is meaningful in all imperative (Pascal-like) languages with updateable variables. It does not apply to Prolog, where addresses never available (in particular, there is no assignment!).

```
append([a], [b], Result)
           in    in     out
append(First, Second, [a,b])
            out       out        in
append([X,b], [c,Y], [a,P,Q,d])
```

none of the above!

Pass-by-Name (delayed evaluation)

The actual parameter *replaces* the formal parameter: imagine modifying the subprogram body by inserting the actual parameters instead of the formal parameters.

Continuation of the `SWAP` example

MODE = name:

`J` = `6`, `A[3]` = `6`

In the modified body of `SWAP` we have:

```
TMP := J; J := A[J]; A[J] := TMP;
```

We evaluate the address of `A[J]` with a new value of `J`, so we update `A[6]` instead of `A[3]`.

If the location of a parameter is needed (e.g., for assignment), it is re-evaluated every time. [In such a case, it must be a variable, perhaps subscripted.]

If the value of a parameter is needed, it is also evaluated every time it appears in the modified body of the subprogram.

Delayed evaluation of an expression requires access to its variables (environment).

```
program PP;
  var X, Y: integer;
  procedure P(name X: integer);
    var Y: integer;
  begin {...}
        write(X);
        {...}
  end;
begin {...}
      P(X+Y);
      {...}
end.
```

Which `X` and which `Y` should be accessed? In the body of `P` the names `X`, `Y` are local (a parameter, a local variable). The modified body of `P` will contain the call `write(X+Y)` but the *call* to P in the main program knows only the global `X` and `Y`. The expression `X+Y` must be passed around with the appropriate environment for its variables.

Jensen’s device

```
function SUM_UP(
          name EXPR: real;
          name I: integer;
          in N: integer) return integer is
  SUM: real;
begin
    SUM := 0.0;  I := 1;
    while I  ≤  N loop
        SUM := SUM + EXPR;
        I := I + 1;
    end loop;
    return SUM;
end;
```

```
X := SUM( 2.0, K, 50 );
   X = 2.0 + ... + 2.0 = 100.0
Y := SUM( A[J], J, 4 );
   Y = A[1] + A[2] + A[3] + A[4]
T := SUM( 2*M - 1, M, 5 );
   T = 1.0 + 3.0 + 5.0 + 7.0 + 9.0 = 25.0
```

Functions

A function produces *one* value, returned by a statement such as

`return` *expr_for_function_value*`;`

or assigned to the function name (as in Pascal).

When functions are used in expressions, they enrich the language by, in effect, introducing new operators. For example, if we define `EXP(X, Y)`, a function for X^Y, we can write

```
Z := 2.0 * EXP(A, N);
```

even if exponentiation is not supported by our programming language.

Side-effects in functions are a problem. Ideally, only `in` parameters (and no side effects) should be allowed in a mathematically correct function. If side-effects are really needed, a procedure should be used—with one parameter set aside to transmit the value that a function would compute.

Design issue (orthogonality!): restricting types of objects that can be returned by functions.

Subprograms as parameters

The concept is simple: an algorithm that depends on an embedded algorithm. A typical example is integration—finding the area under a curve described by a function.

```
function DEFINITE_INTEGRAL(
      function F(ARG: real): real;
      LOW, HIGH: real): real;
const N_INTERVALS = 1000;
var DELTA, SUM, X: real; I: integer;
begin
   DELTA := (HIGH - LOW) / N_INTERVALS;
   X := LOW;
   SUM := F(X);
   for I := 1 to N_INTERVALS - 1 do
   begin
      X := X + DELTA;
      SUM := SUM + 2.0 * F(X)
   end;
   X := X + DELTA;
   SUM := SUM + F(X);
   DEFINITE_INTEGRAL := 0.5 * DELTA * SUM
end;
```

Any `real`→`real` function can be passed to this "metafunction":

```
function SQ( A: real ): real;
begin SQ := sqr(A) end;
{...}
writeln(DEFINITE_INTEGRAL(SQ, 0.0, 9.0));
```

This produces (approximately!) `243.0`.

Overloaded subprograms

This is similar to overloading for operators (see p. 93 of the notes): a procedure or function can be defined many times with the same name, so long as the types of all parameters are not the same. (This is allowed in Ada and in C++.) Example:

```
function ADD1(A: integer) return integer;
begin return A + 1; end;
function ADD1(A: date) return date;
begin if not_last_in_month(A) then
         A.day := A.day + 1;  return A;
      elsif -- ......
end;
```

[skipping: Sections 8.8, 8.9 of the textbook]

Summary

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....