Principles

MacLennan's Principles

- **Abstraction**
  - Avoid requiring something to be stated more than once; factor out the recurring pattern.
  - Subprograms, user defined types, inheritance
- **Automation**
  - Automate mechanical, tedious, or error-prone activities.
  - Garbage collection; looping structures

MacLennan’s Principles (2)

- **Defense in Depth**
  - Have a series of defenses so that if an error isn't caught by one, it will probably be caught by another.
  - Array bound being part of type; definite loops.
- **Information Hiding**
  - The language should permit modules to be designed so that (1) the user has all of the information needed to use the module correctly, and nothing more; and (2) the implementor has all of the information needed to implement the module correctly, and nothing more.
  - Modules, packages, objects

MacLennan’s Principles (3)

- **Labeling**
  - Avoid arbitrary sequences more than a few items long. Do not require the user to know the absolute position of an item in a list. Instead, associate a meaningful label with each item and allow the items to occur in any order.
  - Case statement, position-independent parameters.
- **Localized Cost**
  - Users should only pay for what they use; avoid distributed costs.
  - Violations: Algol60 loops, dynamic type binding.

MacLennan's Principles (4)

- **Manifest Interface**
  - All interfaces should be apparent (manifest) in the syntax.
  - Module specifications; function prototypes
- **Orthogonality**
  - Independent functions should be controlled by independent mechanisms.
  - Algol68 types; Ada parameter passing

MacLennan's Principles (5)

- **Portability**
  - Avoid features or facilities that are dependent on a particular machine or a small class of machines.
  - Ada prohibition of aliasing; C/Algol60 I/O
- **Preservation of Information**
  - The language should allow the representation of information that the user might know and that the compiler might need.
  - Definite looping structures

MacLennan’s Principles (6)

- **Regularity**
  - Regular rules, without exception, are easier to learn, use describe and implement.
  - Violations: strings in most langs; Pascal functions.
- **Security**
  - No program that violates the definition of the language, or its own intended structure, should escape detection.
  - Strong typing in Algol60, Pascal, Ada, C++

MacLennan’s Principles (7)

- **Simplicity**
  - A language should be as simple as possible. There should be a minimum number of concepts, with simple rules for their combination.
  - Pascal; Ada name equivalence
- **Structure**
  - The static structure of the program should correspond in a simple way to the dynamic structure of the corresponding computations.
  - Single entry / single exit; violation: Pascal’s upside-down procedures first form.

MacLennan’s Principles (8)

- **Syntactic Consistency**
  - Similar things should look similar; different things different.
  - Violations: ,/; in Algol68; blocks vs compound statements; Ada private / limited private
- **Zero-One-Infinity**
  - The only reasonable numbers are zero, one, and infinity.
  - Array dimensions; identifier sizes; function nesting

More Principles (Ghezzi & Jazayeri)

- **Efficiency**
  - translation & execution
- **Readability**
- **Writability**
- **Reliability**
  - rigorous semantics; clear distinction between static and dynamic checks; modularity

More Principles

- **Predictability**
- **Learnability**
- **Maintainability**
- **Verifiability**
- **??? Must it end in “ility”?**

More Principles (Yemini & Berry)

- **Wide horizon:**
  - Whenever the semantics of a construct, C, in a language for concurrent programming implies the delay of a process (task) executing in C, C should be able to have other alternatives, and all such constructs should be able to serve as alternatives to each other.
  - Not satisfied by semaphores, monitor calls or Ada select statement.

More Principles (Yemini & Berry) (2)

- **Closure under binding:**
  - For every concurrent program, there exists a sequential, possibly non-deterministic, program with an equivalent semantics.
  - No unitask program in Ada can simulate:

```
declare
   task t1 is  s1.e1()  end t1;
   task t2 is  s2.e2()  end t2;
 begin   end;
   -- s1 and s2 are called in arbitrary order.
```

What's this violate?

```
 COMMON a, b, lab
   . . .
3  WRITE(6,5)
5  FORMAT( . . .)
   ASSIGN 3 TO lab
   CALL sub2
   END

   SUBROUTINE sub2
   COMMON c, d, lab
    . . .
   GOTO lab
3  i = 1/0
   RETURN
   END
```

What’s this violate?

```
10 i=0
   call doThing(... i ...)
   i=i+1
   IF i <= n THEN GOTO 10
```

What’s this violate?

```
i = 10;
s = 10;
 . . .
s = "this is a string";
 . . .
IF i != 10 THEN someFunc(i);
 . . .
s = 3.14159;
 . . .
IF s = pi THEN . . .
```

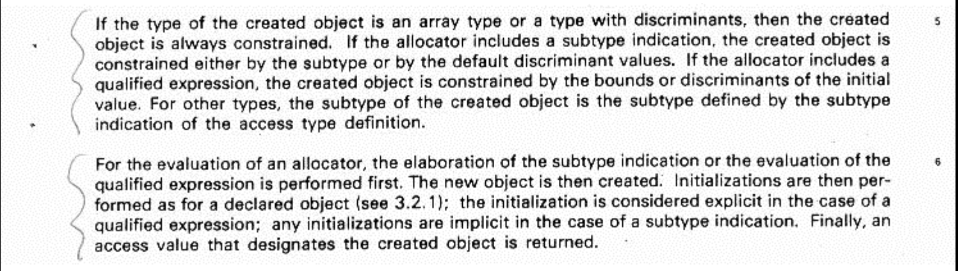
What’s this violate?

```
val = -7
a= val >> 2
val = 15
a = val << 18
```

What’s this violate?

```
GOTO i-- branch to i-th statement
   <statement 1>
   <statement 2>
    . . .
   <statement i>
    . . .
   <statement n>
```

What's this violate?



If the type of the created object is an array type or a type with discriminants, then the created object is always constrained. If the allocator includes a subtype indication, the created object is constrained either by the subtype or by the default discriminant values. If the allocator includes a qualified expression, the created object is constrained by the bounds or discriminants of the initial value. For other types, the subtype of the created object is the subtype defined by the subtype indication of the access type definition. 5

For the evaluation of an allocator, the elaboration of the subtype indication or the evaluation of the qualified expression is performed first. The new object is then created. Initializations are then performed as for a declared object (see 3.2.1); the initialization is considered explicit in the case of a qualified expression; any initializations are implicit in the case of a subtype indication. Finally, an access value that designates the created object is returned. 6

What's this violate?

```
DIMENSION a(20)
CALL subr(a)
 . . .
END

SUBROUTINE subr (n)
DIMENSION n(25);
 . . .
IF n(22) = ...
 . . .
END
```

What’s this violate?

```
i:= 1;
WHILE i < n DO
  BEGIN
    WRITE ( exp(i));
    i:= i+1
  END;

i:= 1;
REPEAT
  WRITE ( exp(i));
  i:= i+1
UNTIL i = n

FOR i:= 1 TO n-1 DO
  WRITE ( exp(i))
```

What’s this violate?

```
FOR (; curr != NULL ; curr = temp)
{
  temp = curr->next;
  FREE ((char *) curr);
}
```

What’s this violate?

```
IF ( I = 0 ) …
IF ( I == 0 ) ...
```

What’s this violate?

```
10  IF (n > 0) THEN GOTO 20
    n = 0
    GOTO 22
20  IF (n > 0) THEN GOTO 30
    IF (m > 0) Then GOTO 25
    n = infinity
    GOTO 40
25  n = m/2
30  IF (n < 1000) THEN GOTO 40
    n = n - 1729
    GOTO 10
40  CONTINUE
```

What’s this violate?

```
GOTO 10
 . . .
ASSIGN 20 to n
GOTO n, (10,20,30,40)  -- assigned GOTO
 . . .
n = 3
GOTO (10,20,30,40), n   -- computed GOTO
```

What’s this violate?

```
WHILE inch = ' ' DO
  READ(infile, inch);
i:= 1;
WHILE inch <> ' ' DO
  BEGIN
    READ(infile, inch);    scannedText[i]:= inch;
    i:= i+1
  END
WHILE inch = ' ' DO
  READ(infile, inch);
```

What’s this violate?

```
I++
++I
I+=1
I=I+1
```

What’s this violate?

```
TYPE stackType = array[1..100] of INTEGER;
VAR stack: stackType;
    top: integer = 0;
PROCEDURE PUSH (object: INTEGER);
 . . .
END; {PUSH}
PROCEDURE POP (VAR object: INETGER);
 . . .
END; {POP}
 . . .
BEGIN {main}
  push(10);
  IF stack[1] = . . .
   . . .
END. {main}
```

What’s this violate?

```
PRINTF("Address of x: %d\n", &x);  -- Fails sometimes

PRINTF("Address of x: %ld\n", (long) &x);  -- Succeeds
```

What’s this violate?

```
VAR i: INTEGER;

PROCEDURE ref (VAR j: INETGER);
  VAR x: INTEGER;
  BEGIN
    j:= 2;
    x:= j+i;
    WRITELN (x)
  END; {ref}

BEGIN
  i:= 1;
  ref(i);
  . . .
END.
```

What's this violate?

```
TYPE stackType=ARRAY[0..100] OF INTEGER;
     stptr=^stackType;
  . . .
FUNCTION f0 (x: INTEGER; y:BOOLEAN): stptr;      -- OK!
FUNCTION f1 (x: INTEGER; y: BOOLEAN): stackType  -- ERROR!
```

What's this violate?

```
TYPE PiType = (tenn, others);
     PiRec = RECORD
               CASE PiType OF
                 tenn: (intPi: integer);
                 others: (realPi: real);
VAR wholePi: Pirec;
BEGIN
  wholePi.intPi:= 3;
   . . .
  WRITELN(wholePi.realPi . . .)
END.
```

What’s this violate?

```
    IF n = 0 THEN GOTO 20
    print("n is not zero")
    GOTO 30
20  print("n is zero")
30  CONTINUE
```