• **ALGOL’s Major Contributions (new language features):**

1- **Free format** (no FORTRAN restrictions!).

2- **Block structuring** the code, introducing the following structuring tools:
   i) “Blocks” and “compound statements”.

   ii) Powerful structuring constructs: “switch”, “for”, nested “if”s, conditional expressions. (look point “8” below)

3- **“Stack” model of computation which facilitates the following new features:**

   i) **Recursion** (power vs. speed/readability).

   ii) **Dynamic Arrays**: array dimensions’ values might be dynamic (known at run-time)

   iii) **Dynamic binding of names to memory spaces**, at run time, due to the stack model of computation → dynamic procedures/functions calls, recursion! (static type binding of names still holds).

   Question: Why does the stack model of computation enforce dynamic binding of names to memory locations, and not names to their types?

   Question: what would enforce dynamic binding of names to their values, at run time?

   iv) **Nested scopes**.

   Question: Why do not we just write M, sub1, sub2, … at the same level instead of nesting sub1 and sub2 within M nesting, i.e., why nesting subs’ abstractions?

4- **ALGOL is a more secure language than FORTRAN**, yet it still keeps the efficient static coping and type checking! Algol has eliminated most of the FORTRAN features that led to “Security Loopholes”, e.g., implicit type declaration, ignoring blanks, COMMON/EQIVALENCE, overworking integers. Unfortunately, while aiming at polymorphic power introduced a “security loophole” (see pass by-name).
5- **Static Scoping**: ALGOL is statically scoped.

- **“Static Scoping”**: The meaning of a name is interpreted according to the static (lexical) structure of the its hosting program module. For example, a non-local variable name “X” which is defined in module M\textsubscript{use} will have its meaning (type declaration binding) from the environment of M\textsubscript{use}’s defining module, say M\textsubscript{def}, according to the static contour diagram of the program, regardless of the caller of M\textsubscript{use}.

- **“Dynamic Scoping”**: The meaning of a name is interpreted according to the run time dynamic behavior (calling sequence) of the its hosting module, M\textsubscript{use}. A non-local name in M\textsubscript{use} gets its meaning from the environment of the caller of M\textsubscript{use}, and not the environment of its definer module M\textsubscript{define}. Hence, the contour diagram is of no use in case of dynamic scoping! Moreover, it forces the inefficient dynamic type checking. Yet, its major advantage is the “power” of polymorphism, where it facilitates the manufacturing of generic polymorphic modules, where the same code is interpreted differently at run time according to their callers.

**Question**: Can we draw a static name visibility “contour diagram” for the dynamically scoped languages?? Justify your “yes/no” answer!

**Question**: In which special case(s) the dynamic and static scoping policies will work the same, i.e., no difference? (Hint: two cases)

<table>
<thead>
<tr>
<th>Feature</th>
<th>“Static Scoping”</th>
<th>VS. “Dynamic Scoping”</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) The code static structure reflects the run-time behavior.</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>2) Does it enforce of run-time type checking?</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>3) Is it more efficient?</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>
4) Is it more powerful? ?? ??

5) Secure? ?? ??

- **Example of Static vs. Dynamic Scoping:**

```plaintext
Program TEST;
    var a, b: integer;
    c: real;
Procedure P1(var x: integer);
    var a: real;
Procedure P11(var y: real);
    var r: real;
    begin r := y + a; writeln (r); end;
Procedure P12(var z: integer);
    var a: integer;
    begin(* P12*)
        a := 20;  P11(1.5);
    end(*P12*)
begin (* P1*)
    a := 2.5;
    P12(x);
end;
begin (* TEST*)
    a := 15;  b := 20;  c := 1.7;  P1(a);
end (*TEST*)
```

6- **Parameter passing:** i) “by-value” (user view of “input” parameter) and ii) the very powerful “by-name” (default, input/output parameter).

**Pass “by-value”:** The value of the actual parameter, at the caller side, is placed in its corresponding formal, the callee’s AR. For the first time we can say that the *user view* is considered when we think of by-value parameter is an “*input*” parameter. Hence, now the compiler can guard against misuse of the input parameter, e.g., when used as an output parameter (l-value), by the programmer. If such protection exists, the compiler, for efficiency considerations, can implement the passing of
value or reference (internally) for scalars and composite structures, respectively.

Pass "by-name": The compiler generates a machine code, function like, called "thunk" for every actual parameter, at the caller side, instead of carrying out the calculation of the final value of the actual. The thunk will range from just a very simple single reference (address), in case of a simple variable name actual parameter, to a very complicated code of an expression actual parameter involving many references of all involved names in the expression. All references in the thunk will point to the caller’s AR slots, specifically to where the involved names in the actual parameter expression. You can always think of pass by-name as textually substituting the formal parameter by an exact textual copy of its corresponding actual parameter everywhere in the callee’s code. Hence, it is a very powerful mechanism since the thunk can be a very complex construct (e.g., a function call), where its evaluation is delayed until the evaluation of its formal parameter (lazy evaluation). If any of the involved names in the thunk’s code changes, the next evaluation of the formal parameter will be different, which will result in a different value from its last invocation (polymorphic power).

“by-name” is very powerful (see the polymorphic Jensen’s device page 131), but also dangerous (see the swap example page 133).

Question: Does by-name facilitate passing a “function” as a parameter?

Question: When would it be the case where by-name and by-reference are the same? [Think the nature of the actual parameter]

7- “0-1-∞” design principle: (page 117)
For any introduced feature in the language, you do not ask the users to remember any specific restricting numbers; or if you must, it should be either 1 or any number.
For example, in ALGOL (theoretically) there is no limit on the label length and block/procedure nesting depth.