1- **Parameter passing**: i) “by-value” (user view of input parameter) and ii) the very powerful “by-name” (is the 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 of the actual parameter]
2- **“0-1-∞” design principle:** (page 117)
For any language feature, the users do not have to remember any specific restricting numbers; or if they 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.

3- **Generality of Control Structures:** (page 121)
- Extending the “if” statement of FORTRAN to “if-then-else”.
- Extending the “DO-loop” statement of FORTRAN to “for” statement
- Definite looping: \( \text{for } i := 1 \text{ step } 2 \text{ until } N * M \text{ do } \) statement;
- Indefinite looping: \( \text{for } \text{NewGuess := Improve (OldGuess)} \) 
  \( \text{while } \text{abs}(\text{NewGuess} – \text{OldGuess}) > 0.0001 \text{ do } \) statement;

  In my view, it is a bit confusing syntax; mixing the definite and indefinite semantics for looping!

Here is another very powerful “for”! (I do not think any other languages would have a more general one!): (top of page 138)

\[
\text{for } i := 3, 7, \\
11 \text{ step } 1 \text{ until } 16, \\
i/2 \text{ while } i \geq 1, \\
2 \text{ step } i \text{ until } 32 \\
\text{do } \text{print } (i)
\]

the output (values of i) is:
\[
3 \quad 7 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \quad 8 \quad 4 \quad 2 \quad 1 \quad 2 \quad 4 \quad 8 \quad 16 \quad 32
\]

- The selections statement “switch”:
  \[
  \text{begin } \\
  \text{switch } S = L , \text{if } i > 0 \text{ then } M \text{ else } N , \text{ Q } ; \\
  (\text{evaluate } i, j) \\
  \text{goto } S[j]; \quad (* \text{if } j=2 \text{ then the value of } i \text{ will decide jumping to either } M \text{ or } N *)
  \]

  L: **** 
  \text{goto } \text{done}

  M: ****
  \text{goto } \text{done}

  N: ****
  \text{goto } \text{done}

  Q: ****

  \text{done:}
  \text{end}
* Notice that the “switch” and “for” statements are “baroque”.

*ALGOL solved the dangling “else” problem (DE):

If B then if C then S else T;
Does the “else” relates to the first or second “then”

**Solution:** ALGOL restricted the consequent of the “if” must not be another if statement:
Hence it is illegal to write “if A then if B then D else C

* Remember that the lack of defining reserved words in FORTRAN was a major factor that resulted in a security loophole.

Three Lexical conventions for words:

**Reserved words:** (most modern languages) reserved for the language use, can not be used by the user as ids!
Ex: if, procedure, begin, end, while, for, procedure, function, …

**Keywords:** (ALGOL approach) used by the language and unambiguously marked to be used by the language, yet the programmer can use them for id’s if they are not marked! (marking: boldface, preceded with #, surrounded by quotes)

**Keyword in context:** (FORTRAN/PL-I) words are keywords only when expected in their context:

EX: IF IF THEN
    THEN = 0
    ELSE
      ELSE = 0
BNF Grammar (Backus-Naur Form)

Meta-language that describes another language’s syntax.

It is equivalent to context free grammar (CFG) :  ( {T}, {N}, {P}, S) where
{\{T\}}: set of all terminal symbols: 0,1,2,…, a, b, c, …, +, -, …,
Symbols that can not be reduced further more.
{\{N\}}: set of all nonterminal symbols: statement-sequence, if-statement, expression, ....
Symbols that need to be further reduced (replaced with other expansion symbols) (expanded).
{\{P\}}: set of production rules,
the left hand side symbol must be a “nonterminal”

Ex:  <unsigned integer> ::=  <digit> |  <unsigned integer>

S: starting symbol (S belongs to {N})

Ex:  <program> ::=  program <header> ;  
     <declaration-section> ; <program-body> end

<program> is the starting symbol in this case.

A regular grammar is either a left or right grammar.
A right regular grammar is same as CFG, but all production rules P are one of the following rules:
1- A \rightarrow a -  A is a non-terminal in N and a is terminal in T
2- A \rightarrow a B -  A and B are non-terminal in N and a is terminal in T
3- A \rightarrow \epsilon (empty string) -  A is a non-terminal in N

A left regular grammar is same as above except for rule 2,
where “A \rightarrow Ba” replaces of A\rightarrow aB

A regular grammar can be both, right and left grammar, otherwise it would be CFG.

The BNF is powerful enough to describe the following syntactic issues in a programming language definition:
1- Lists of similar constructs: statement-sequence, declaration-sequence,…
2- The order in which different constructs must appear: a label must start with a letter not a digit.
3- Nested structures to any depth: nested statements
4- Matching parentheses: ((((A+B))))
5- Operator precedence: the / has higher precedence over the +
6-  Operator associativity:

BNF for expressions:

<expr> ::= <expr> <add-op> <term>                (* left associative BNF*)
  |  <term>
<term>  ::= <term> <mult-op> <factor>      (*precedence: mult-op > add-op*)
  |  <factor>
<factor> ::= <id>  | <number>  | - <factor> | (<expr>)

<add-op> ::= − | +

<mult-op> ::= / | *

In general, there are two way to evaluate this assignment: X := A − B + C

i) If the BNF expresses left associative then the above is execute as
   X := (A − B) + C
   Left associative: <expr> ::= <expr> <add-op> <term>

ii) But in case of right associative then the above is execute as
   X := A − (B + C) ;
   Right associative: <expr> ::= < term> <add-op> <expr>

Yet, BNF can never describe any semantics (or context sensitive issues! (e.g., variable not declared, over/underflow, label length is not acceptable, …).