Topics:

1- The position of High Level Languages (HLLs) in the computer system (core hardware, micro-level, macro-level (assembly), OS/HLLs).
   HLLs allow us in the feasibility to utilize the hardware (black-box) for the implementation of toughest algorithmic solutions of most problems that we face.

2- Why Study HLLs?
   i) Better understanding of their features, allowing us, when the need arises, to have a smart choice of the best HLL for our algorithmic solution implementation!

   ii) Efficient improvement of existing HLLs to fit better our needs.

   iii) Future design of new HLLs.

3- What makes a “good” HLL?
   a) Clarity of its syntax and semantics.
   b) Richness and independency of its features and constructs that makes it easy to find suitable one, mix and combine many, for more efficient software implementation.
   c) Its support of abstraction: i) user defined ii) built-in.
   d) Its support of security at: i) development of software ii) run time robustness
   e) Program portability between different platforms.
   f) The cost of program: i) development ii) translation iii) maintenance
   g) Its support of useful software development environment (to the user), e.g., editors, interpreters, graphical user interface.

4- What are the major factors that characterize a HLL?
   a) Power (e.g., short syntax that reflects powerful semantics, the programmer can do anything he/she dreams to do! recursion, polymorphism).
   b) Abstraction and modularity: it helps at all phases of SW design, code reusability, easy maintenance, testing, separate module design.
   c) Security: language & applications.
   d) Program’s speed of execution.
   e) Readability and how easy to get the semantics from its corresponding syntax.
In the design of a general purpose HLL, our main challenge is to find the optimal point between the above *contradicting* factors!

In case of the design of a special purpose language, we tailor the language based on what we need for specific application, where some of the above factors might turn obsolete.

**High Level Languages Paradigms:**

A) *Imperative:* Action oriented, the programmer dictates to the CPU how to execute the code via *sequence of commands* (instructions), where the execution control flow is govern by an instruction counter, and possibly, changing the computer state with every instruction’s execution.

Example HLLs: FORTRAN, PASCAL, BASIC, C, Ada, Modula, C++, Smalltalk,…

i) Block-Structured (non-object oriented). Data are passive and abstraction is weak and artificially added.

ii) Object-Oriented HLLs: Data are active, they have behavior, acting on themselves and the other data in the system, via a message based mechanism. Abstraction is inherent in the language; every datum is an object ADT. For code sharing the concept of reusability is introduced via the *inheritance* mechanism.

B) *Declarative:* Defining the problem to solve via a set of function calls or rules of inference, hence there is now “how” to do it CPU, and no intermediate changing of the system memory values (i.e., state)!

---Tue. 1.24.2006---

i) Functional: * A program is one function composition call.
  * The equivalency of programs and data (data and programs are lists).
  * No intermediate memory side effect (change of system state).
  * Recursion replaces iteration.

Examples: Lisp, ML, Scheme, Miranda, id, Haskell

ii) Logic: The program is a set of axioms and rules that describes the programming environment; then the system evaluate the assertion of a “goal” (theorem”). The program output(s) are obtained as a side effect of the goal evaluation. It is the most abstract domain, “what is the problem to solve” replaces “how to solve it”.

---Tue. 1.24.2006---
HLLs Translation and Software Simulation (HLL Virtual Machine Interpreters)

A) HLLs Translators:

i) **Compilers:**

HLL program \( \rightarrow \) **Scanner** \( \rightarrow \) Token stream \( \rightarrow \) **Syntactic Analysis** \( \rightarrow \) Abstract syntax parse tree \( \rightarrow \) **Semantic Analysis** \( \rightarrow \) Abstract syntax parse tree \( \rightarrow \) **Optimization** \( \rightarrow \) **Code Generation** \( \rightarrow \) Machine code

**Scanner:** scans the input program statements and extracts its composing tokens.

Example: \( Y := X + Z ; \)
Output tokens: \( Y, :=, X, +, Z, ; \)

**Syntax Analyzer:** develops an abstract syntax parse tree (AST) detecting syntax errors.

**Semantic Analyzer:** takes the above AST and parse it for semantic consistency (e.g., consistent use of operators and data types).

**Optimization:** optimizes the parsed tree for efficient code generation.

**Code Generation:** uses the resultant abstract representation for generating executable machine code.

ii) **Assemblers:** Assembly program \( \rightarrow \) Assembler \( \rightarrow \) executable machine code

ii) **Preprocessors:** extended HLL program (C++) \( \rightarrow \) **Preprocessor** \( \rightarrow \) HLL code (C)

B) Software simulators (Interpreters):

HLL program \( \rightarrow \) **The HLL’s Virtual Machine** \( \rightarrow \) program output

Examples: Java, Lisp, ML, Smalltalk, Prolog.
FORTRAN (OLD VERSIONS) [Backus 1956, IBM]

- First Widely used HLL.
- The main design goals: 1) Efficiency  2) Amenability for Numerical Engineering Applications.
- All needed storage of the program is decided at compile time, minimal invocation of the OS.
- Simple typing system, static, and few types: integer, real, complex, double, and arrays (new versions added: Boolean, character string, and file).
- No type definition or facilitating user defined data abstractions, only arrays and fixed size char string. In addition, Subroutines and functions.
- Parameter passing by “reference” or “value-result”.
- FORTRAN programs are divided into disjoint subprograms’ environments, and a main program.
- Program structure: declarative and imperative sections-
  1) Declarative section serving the following functions:
     i) Allocating memory spaces of specified sizes (based on the name declared type) to hold
        the declared names, bind them statically (in FORTRAN) for the entire life cycle of the
        program execution.
     ii) Possibly, assigning initial value (if given) to the name, in its allocated memory space.
  2) Imperative section holds one of the following statement types:
     i) Computational:   \( X = \frac{Y}{Z} + F(4) \)
     ii) Control Flow:   \text{GO TO, DO .. CONTINUE, IF (L1, L2, L3) I, CALL SUB1(X,Y,Z),…}
     iii) Input Output: READ, PRINT

2.2 Control Structure

- The “GO TO” is a simple, yet very powerful, known to be the workhorse of the control flow. It led to formation of a “bad” control structure, violating the structure principle (p. 49).
- There are three types of the “GO TO”:
  i) Unconditional:   \text{GO TO label}
  ii) Computed:    \text{GO TO (L1, L2, …, Ln), I}
                   If \( I = k \), jump to \( L_k \) label in the label list, \( 1 \leq k \leq n \); otherwise no jump.
  iii) Assigned:   \text{GO TO N, (L1, L2, …, Ln)} \quad [ \text{the label list is not used by the compiler!!!}]
                   go to the address contained in \( N \), \( N \) must be pre-assigned some label address, via
                   the assign statement \text{ASSIGN label TO N}, that places the address of label into \( N \).
                   It is the responsibility of the programmer to do so (leads to insecurity).

Language’s Security Loophole: The similarity of the ii) and iii), the overworking of the integer type to carry label’s address and integers (weak typing), and trusting the user to use the assign statement before writing iii) would introduce a great possibility of the CPU jumping to execute at an unknown place in memory (if we are lucky, we get segmentation violation, otherwise we get what seems “good” program result, which is not)
A Typing System

- **What is a “Type”**? A type of a variable is the set of values that such variable can have and the set of operations that can work on such values.

- **Why do we need Types in some HLLs?** Mainly for the following reasons:
  1) efficient allocation of memory
  2) Type checking

- There are two major type classifications:
  1) **built-in** system: basic types-- integer, real, etc and structure—array, records, files
  2) **user defined**: ADTs, Classes, Modules, Packets

- Type “Coercion”: implicit type conversion based on the context of use.

\[
\begin{align*}
X + I & \rightarrow X + \text{FLOAT}(I) \\
X = I & \rightarrow X = \text{FLOAT}(I) \\
I = X & \rightarrow I = \text{IFIX}(X)
\end{align*}
\]

- **Explicit vs. Implicit Declarations**:

  FORTRAN has both implicit (language convention: explicit declarations (via a declaration statement), if not then all names starting with “I through N” are considered integers, otherwise they are reals. Implicit declarations are power in the language, but it could easily lead to a language’s security loophole as we will see later, when the programmer when the compiler considers a misspelled name to be a valid name and use its incorrect value instead of the correct intended name’s value.

- **Strong vs. Weak typing system**:

  A “strong” typing system has to maintain the following:

  i) A rich enough set of types not to overwork any of its types to represent more than one set of values, e.g., FORTRAN’s weak typing system overworked the type integer by having it also to carry labels’ addresses, and strings (Hollerith strings are considered integers)! The programmer can make a silly mistakes without the system being able to detect them; i.e., \(N = \text{ISUCC}(6HCARAMEL)\).

  Ex: Pascal’s introduction of the “**enumerated**” types!

  ii) A type checking mechanism that guarantees the correct and safe manipulation of data at all program constructs, e.g., formal actual parameters type/count matching, operands type matching in all expressions.

  Such type checking might be:

  1) Static: at compile time for efficiency.

  2) Dynamic: at run time for polymorphic power.
iii) No user trust of doing the safe/correct thing!

- A secure language has to have a strong typing system, and more!

In addition to having a strong typing system, for a HLL to be secure, other language definitions (that aim at other desirable features, e.g., power, abstraction, etc) have to be secure and not leading to undetectable errors, e.g., by-name in Algol, and dynamic genericity in Modula-2 (both are powerful features).

A language with a weak typing system will overwork some types, and “trusts” the user to use them wisely (securely), hence the language is insecure; because it is guaranteed that the user will make undetectable errors, neither at compile or run times.

Parameter Passing Modes in FORTRAN

Call “by-reference” parameter passing only (until FORTRAN 77)

- If the actual parameter is an l-value, e.g. a variable, its reference is passed to the subroutine
- If the actual parameter is an r-value, e.g. an expression, it is evaluated and assigned to an invisible temporary variable whose reference is passed to the corresponding actual parameter. In early versions of FORTRAN, all constant numbers (e.g., 2, 45, 100, 3, …) were stored in the literal table and their addresses in memory are known at compile time. Such addresses would be passed to the corresponding actual!! And, guess what, if we mistakenly change any formal, it corresponding actual would change at the caller side. This is a clear case of security loophole that does not relate to the typing system weakness/strength!

- For example: In a hypothetical FORTRAN-II coding with pass by-reference:

```fortran
SUBROUTINE ALTERNUMERIC(A)
    INTEGER A
    A = 99
    RETURN
END
```

At the main program:

```fortran
CALL ALTERNUMERIC(7)
J = 5
J = I + 7   actually the number 7 is now has 99 on its allocated memory slot!
PRINT ( J)   the output is 104  !!!!!!! Such error is undetectable (security loophole)
```
Pros: 1- Efficient when passing huge structure (arrays)
2- the power of aliasing (if you do not care about security!!)

Cons: 1- The lack of the clear distinction, by the language, of input versus output parameters might easily lead to making a meaningless operations, like storing values into expressions and changing literal constants. Hence, the compiler can not stop any value assignment to a constant/expression actual parameter, since all the compiler sees is a formal parameter name, that might correspond to a constant (right-value) or name (left-value) actual parameter. !!!!!!!!!!!!SECURITY LOOPHOLE!!!!!!!!!

2- In modern FORTRAN versions, “by-reference” and “global name access” could lead to aliasing two or more names on the same memory location.
Example:

```fortran
SUBROUTINE ALIAS(A, B, C, Z)
INTEGER A, B, C, Z   (remember there is an implicit visibility of S from now on)

(Now, we also have five names pointing, aliasing, into the same allocated slot for S, at the caller side , namely: A, B, C, and S, in main program)

A = 10
B = 20
C = 30
S = 400
Z = A + B + C
RETURN
END
```

At the main program code, if we have a name S to be global, i.e., visible everywhere:

```fortran
INTEGER S, H
CALL ALIAS (S, S, S, H)
PRINT (S)
```

the output is not 60 as we expect; instead it is 1200 (why!)

Call “by-value-result” parameter passing frequently used after FORTRAN 77

- CALL BY VALUE-RESULT (Pass-By-Value-Result):
  - the value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable.
  - at subprogram termination, the value of the formal parameter is transmitted back to the actual parameter.
- Disadvantages:
  1- requires multiple storage for parameters and the time for copying values.
  2- can be parameter collision: if the procedure is called with the same variable as more than one actual parameter. The order that the formal parameters are copied back to the actual parameters (at the end of the subprogram) determines the value of the variable.

Look Exercise 2-15 and 2-16 p. 60-61 in your text.
Subprograms Are Implemented Using Activation Records (AR):

When changing the control of execution from the caller to the callee (upon encountering a subroutine CALL statement), we need to save the “state” of the caller in some data structure (e.g., record), that is called “Activation Record”. Such saved state will be used to restore the activation of the caller upon the return from the callee subroutine, i.e., when executing the RETURN statement.

The state of a computation (program in execution) consists of:
1- All of its local&formal parameters (if any), and any temp values in registers (TEMP).
2- The instruction pointer (IP) that points to the next statement to be executed. At the subroutine invocation, the IP will hold the resume (return) address in the caller code.
3- Pointer to the caller’s AR called “dynamic link” (DL).

Steps taken upon subroutine invocation (call):
1- Allocate an AR for the callee.
2- Save the “state” of the caller in its AR.
3- Compute the actual parameters and transfer them to the corresponding formals in the callee AR.
4- Place a pointer to the callee AR into the caller AR (the DL).
5- Transfer the execution to the beginning of the callee code.

Implementation of Non-recursive Call, in FORTRAN:

Code in the caller S:
Call F(P1, P2, P3, …, P n) →

Save Temp values in M[AR(S)].TEMP;
M[AR(S)].IP := resume address (IP);

M[AR(F)].PARM[1] := compute P₁ (address in case of by-reference);
M[AR(F)].PARM[2] := compute P₂ (address in case of by-reference);

***
M[AR(F)].PARM[n] := compute Pₙ (address in case of by-reference);

M[AR(F)].DL := Address of AR(S);

Go to entry (F)

resume:
    Restore registers from M[AR(S)].TEMP;
    If F is a function then get its returned value.
Code in the callee F:
entry (F):

........

RETURN ➔
If F is a function, place the returned value where accessible to the caller;
Go to ( M[AR(F)].DL].IP);

------------------------------------- 2/9/06 ----------------------------------------------

Data Structures:

• Operators are overloaded: it is when an operator is used with integer,
double/float, and complex operands, e.g., the “+” in FORTRAN. It is an ad-hoc
polymorphism.

• Early FORTRAN did not allow mixed mode expressions (of more than one type
operands), instead of implicit coercion, the user must explicitly use conversion,
e.g., I = IFIX (X + FLOAT (I)) when adding real X to integer I. Newer versions
permit mixed mode, running the risk of getting a wrong results if the user is not
careful when writing the expression (oops!, **Security Loophole**). For example,
X^{(1/3)} should return the cube-root of X, instead it returns X^0, since the integer
division 1/3 returns 0. Moreover, we can easily lose precision when assigning
real to integer, e.g., I = X (if X =0.999999999 we still get 0 stored in I).

• In addition of overworking the integer type with labels, it is overworked
with character strings! FORTRAN allows the H-format strings (Hollerith) to
be read integer/real variables (!!) and passed as an actual for a corresponding
integer formal parameter, where we can easily increment a string by one
(**Security Loophole**)!

Name Structure:

As shown before, data structure will structure data, and control structure will
structure the control flow, and “name” will organize all names that are used
in the program.

**Primitives of name structures**: binding constructs: the declaration
statement “INTEGER A, B, C” binds A, B, and C names to the type
INTEGER and the addresses of their corresponding allocated memory slots,
in the static symbol table.

**Environments Determine Names/Constructs Meanings**: 

The context or environment of any language constructs (e.g., statement) is
the visible set of definitions (e.g., declarations) to such construct that gives it
its meaning through the definitions of all of its involved names. For example, \( X = \text{COUNT}(I) \) statement could be interpreted in many different ways/meanings depending on the definition of \( X \), \( \text{COUNT} \) (a function or an array?!), and \( I \).

**Variable Names are Local in Scope:**
The program is divided into disjoint subprograms (environments), for independent abstract implementation. The details of the subprogram concrete implementation, e.g., formals names, local names, etc, are hidden from the user in a separate environment. All the caller knows is the info in the subprogram interface!

In FORTRAN, only subprogram names are global in scope. But, all locally defined names in a subprogram (or the main program) are local in scope to such subprogram; and never visible the outside world.

The *scope* of a “name binding” is the region of the code over which such binding (e.g., type declaration) is visible.

For a better understanding of name structure, we will study the “**contour diagram**” mechanism which visualizes the program modules as boxes; each is made of one-way mirrors that allow inside-out visibility of name binding, but never outside-in!

Look Figs 2.8 and 2.9, pages 79-80

**COMMON Blocks:**

Since there was no global declarations (scoping), except for sub names, FORTRAN facilitated inter-subroutines-communication via global blocks of storage. They thought that communicating via the sub’s interfaces (parameters) would constitute a violation to the sub’s abstraction!

Ex:

```plaintext
SUBROUTINE SUB1( A, B)                    SUBROUTINE SUB2( H, M)
REAL X(100)                                             REAL C(50), D(100)
INTEGER Y(250)                                       INTEGER E(200)
COMMON /BLOCK1/  X, Y                     COMMON /BLOCK1/  C, D, E
***                                                                   ***
END                                                            END
```
Advantages: Better memory utilization via shared memory space, and facilitates inter-subroutines communications, while maintaining subroutines’ abstraction.

Disadvantages: insecure *aliasing* of more than (possibly different types) name to the same space in memory, hence inadvertent storage sharing between different-types names; especially with *no type/count check* for matching, *lack enforcement of re-initialization before use* (!?), and *lack of run-time tagging* the shared memory to know the type of the currently residing value, by the compiler, at different block statements *(Security Loophole)*.

**EQUIVELANCE**: Memory sharing within the same subprogram. Yet, same Security Loophole as in COMMON (*aliasing and no run-time tagging*). It is similar to the “union” in C and variant records in Pascal/Ada.

Both EQUIVELANCE and COMMON are deprecated features to be removed in later versions of FORTRAN (after F90). Yet, they are an example of gaining efficiency & abstraction versus lose of security!

------------------------------------ 2-14-06 --------------------------------------

**Syntactic Structures:**

**Restricted format:** column 1-5 label, 6 for commenting the line, 7-72 for coding statements.

Due to ignoring blanks everywhere and the lack of reserved words in FORTRAN, we would have the following problems *(Security Loopholes):*

1- The user might write “DO20I=1.100” instead of “DO20I=1,100” which would give totally different result when executed!

2- DIMENSION IF(100)
If “IF(I-1)=123” is written instead of “IF(I-1)1,2,3”, such error will not be possibly caught and the (I-1)th element of the array IF will be assigned 123.

ALGOL60 (Naur, 1960)

- Elegancy and generality (powerful universal ALGOritmic Language) are the main design goals of the ALGOL.
- Sample ALGOL60 code: Fig 3.3 page 102.
- 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”.

    Ex. **Block**: `begin declaration-sequence; statement-sequence end`

    They define nested scopes (look example code and its contour diagram pages 102 and 103, respectively.

**Why blocks?**

1- To gain what is meant by COMMON in FORTRAN, yet eliminating the disadvantages! In order to share a common declarations among a number of procedures without being part of their interfaces (hence no chance of inconsistency of different types/names/numbers, or violating abstractions), a block will encapsulate all, allowing for efficient and secure access to such declarations among all using procedures. Thus, blocks aid in the construction of large software. (pages 105-107)

2- They define a separate scope with all declarations, of which there might be a huge arrays (large spaces) that they should not be in the system stack (as part of the callee’s AR) when there is no need for them. Hence, if such arrays exists within some procedures, we simply encapsulate them and their using code only in an internal block. Now the huge array space is part of the block AR and not the procedure AR. Thus when we exit the block, its AR will be deleted from the stack, deal locating the large space AR, and returning to a much smaller procedure AR. On the other hand, if we do not use blocks, the procedure AR would be allocated huge space (due to the large array), and kept in memory while executing the “entire” procedure code. (pages 112-114).
Syntax rules:

statement: simple-statement | compound-statement ;

compound-statement: begin statement-sequence end ;

Why compound statements? To group multiple statements into a single statement to be used wherever it is needed (e.g., if-then-else, for, ...).

Potential problem! If we start with one statement, then we add more later, Unless we compound with “begin” “end” we might have an undetected error.

```
x := 0; y := 1;
for i := 1 to 2 do
  x := x + i;
y := y * x;           (* we added this but did not make the two statement a compound statement, using begin-end)
```

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):
   The power of recursion is stemmed from the math proof by induction!
   Hence, the recursion process solves the main problem, in many steps, each with lesser size input, utilizing the same module code (reusability). The major drawback is the overhead of the extra (machine) code for module invocation/return, which slows down the recursive solutions compared to its corresponding iterative approach. Moreover, sometimes recursive solutions are “a bit” harder to read.

ii) **Dynamic Arrays**, with variable subscript range(s), allowing dynamic array size allocation, at run time (for storage efficiency). Instead of committing to a max size array, we dynamically allocate the exact needed size according to each application.

iii) **Dynamic binding of names to memory spaces**, at run time, due to the new recursion feature! (the static type binding still hold).
   Question: Why does recursion enforces dynamic binding on names to memory locations?

\------------------------------- 2/16/06  \-------------------------------

iv) **Nested scopes**, allowing subprograms nesting, for better abstraction.
    Remember, the scope of all declarations at the most inner box is not visible to any other outer boxes. Whereas, the most outer scope (all declarations in the box that encloses all inner boxes) is visible to all enclosed boxes. When we
nest subprograms (sub1, sub2, …), within a hosting module (say M), we mean to create separate abstractions, i.e., sub1, sub2, …, within M. Hence, any declaration inside sub1 and sub2 should not be visible to the outside of sub1, sub2.

Question: Why do not we just write M, sub1, sub2, … at the same level without nesting, i.e., why nesting abstractions?

4- A more secure language than FORTRAN, yet still efficiently static! Eliminated most of the FORTRAN features that led to “Security Loopholes”, e.g., implicit type declaration, ignoring blanks, COMMON/EQIVALENCE, overworking integers, etc. Yet, the language is not totally secure (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_{use} will have its meaning (type declaration binding) from the environment of M_{use}’s defining module, say M_{def}, according to the static contour diagram of the program, regardless of the caller of M_{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_{use}. A non-local name in M_{use} gets its meaning from the environment of the caller of M_{use}, and not the environment of its definer module M_{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 contour diagram for the dynamically scoped languages?? Justify your “yes/no” answer!
Question: In which case(s) the dynamic and static scoping policies will work the same, i.e., no difference? (Hint: two cases)

- **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*)
```

The output (value of “r”) is **4.0** (1.5+2.5) in case of **static** scoping, and **21.5** (1.5+20) in case of **dynamic** scoping.
6- **Parameter passing:** i) "by-value" (user view of "input" parameter) and ii) the very powerful "by-name" (default, input/output parameter).

**Passing “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.

**Passing “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 single name variable actual parameter, to very complicated code involving many references of all involved names in a complicated 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” copy of the text of its corresponding actual parameter**, everywhere in the callee. 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 its formal parameter invocation (lazy evaluation), which might result in a different value from its last invocation (polymorphic power!).

**Question:** Does by-name facilitate passing a function as a parameter?
Question: When would be the case where by-name and by-reference are the same?

- By-name is very **powerful** (see the Jensen’s device page 131), **but** also **dangerous** (see the swap example page 133).

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.

8- **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:  
  \[ \text{for } i := 1 \text{ step } 2 \text{ until } N \ast M \text{ do statement;} \]
- Indefinite looping:  
  \[ \text{for } \text{NewGuess} := \text{Improve} (\text{OldGuess}) \]
  \[ \quad \text{while } \text{abs} (\text{NewGuess} - \text{OldGuess}) > 0.0001 \text{ do statement;} \]
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)

\[
\text{for } i := 3, 7, \\
11 \text{ step } 1 \text{ until } 16, \\
i/2 \text{ while } i \geq 1 , \\
2 \text{ step } i \text{ until } 32 \\
\text{do print } (i) \\
\]
the output (values of i) is:
3  7  11  12  13  14  15  16  8  4  2  1  2  4  8  16  32

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

M: ****
goto done

N: ****
goto done

Q: ****
done:
end

* Notice that the “switch” and “for” statements are “baroque”.

*ALGOL solved the dangling “else” problem:
The DE problem:
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**: reserved for the language use, can not be used by the user as ids!
Ex: if, procedure, begin, end, …
Used by most modern languages!

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

**Keyword in context**: 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 → a - A is a non-terminal in N and a is terminal in T
2- A → a B - A and B are non-terminal in N and a is terminal in T
3- A → ε (empty string) - A is a non-terminal in N

A left regular grammar is same as above except for rule 2,
where “A → Ba” replaces of A → 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:
    X := A - B + C ;
    If the BNF expresses left associativity then the above is execute as
    X := (A - B) + C
But in case of right associativity then the above is execute as
\[ X := A - (B + C) ; \]
Left associative:  \[ <expr> ::= <expr> + <term> | <expr> - <term> \]
Left associative:  \[ <expr> ::= <term> + <expr> | <term> - <expr> \]

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

**BNF for expressions:**
\[ <expr> ::= <expr> + <term> \]
\[ | \quad <expr> - <term> \]
\[ | \quad <term> \]
\[ <term> ::= <term> * <factor> \]
\[ | \quad <term> / <factor> \]
\[ | \quad <factor> \]
\[ <factor> ::= <id> | <number> | (<expr>) \]

**Extended BNF:**
The same BNF power but it increases the readability and writability of the BNF!
Three extensions are used:

1- “optional” parts in the rhs of a rule:
Ex:  \[ <if-stmt> ::= \textbf{if} (<logic-expr>) \textbf{then} <stmt> [\textbf{else} <stmt>] ; \]

2- zero or more repetitions of some constructs using braces.
Ex:  \[ <stmt-list> ::= <stmt> ; <stmt>\]

3- multiple-choice options:
Ex:  \[ <for-stmt> ::= \textbf{for} <id> := <expr> (\textbf{to|downto}) <expr> \textbf{do} <stmt> \]

**Attribute Grammar to Describe Context Sensitive Languages Aspects:**

\[ <unsigned-int> ::= \quad <digit> \]
\[ \{ \text{value (<unsigned-int>) } \leftarrow \text{value (digit)} \} \]
\[ | \quad <unsigned-int> <digit> \]
\[ \{ \text{Value (<unsigned-int>) } \leftarrow 10 \times \text{Value (<unsigned-int>)} \]
\[ \quad \text{+ value (digit)} \} \]
\[ <digit> ::= 0 \]
\[ \{ \text{Value (<digit>) } \leftarrow 0 \} \]
\[ | 1 \]
\[ \{ \text{Value (<digit>) } \leftarrow 1 \} \]
\[ | 2 \]
\[ \{ \text{Value (<digit>) } \leftarrow 2 \} \]
\[ | 3 \]
\[ \{ \text{Value (<digit>) } \leftarrow 3 \} \]
\[ \quad \text{***} \]
\[ | 9 \]
\[ \{ \text{Value (<digit>) } \leftarrow 1 \} \]
PASCAL

• ALGOL’s like language, but more reliable, efficient, and simple for pedagogic purposes.

• PASCAL introduced a much richer type system than ALGOL:

  A) “type” and “Const” declarations.
  B) “Enumeration Types”: To handle non-numeric data, eliminating the insecurity of overworking integers as such needed types. (exists in C, Pascal, and Ada)

    ```
    type
    month = (Jan, Feb, March, Apr, …, Dec);
    Day    = (Sun, Mon, Tue, …, Sat);
    ```

    **In C**: `enum` StudentClass {Fresh, Soph, Junior, Senior}
    `enum` EmployeeGender {Male, Female}
    **In Ada**: `type` Days is ((Sun, Mon, Tue, …, Sat);

    They are true ADTs, with user defined data type, and system built-in operators: `:=, succ, pred, =, ≠, <, >, ≥, ≤`

    Advantages:
    1- High level application oriented, allowing the language to cover wider area of applications.
    2- Efficient use of memory to represent the type values.
    3- Secure use of type elements, the compiler protects against any meaningless operations by the users on elements of the defined types. Pred(Jan) and succ (Sat) will produce compile time errors.

**Question**: What about the input/output of enumerated types?

Problems? Yes:
1- No Input/output built in operations! (why?)
2- Overloaded enumerated literal constants when appearing in different definitions at the same environment.
   EX: type favoriteColors = (red, yellow, magenta, brown, aqua, blue, green);
       TrafficLightColors = (red, yellow, green);

       **for** color **in** ‘red’ ..’green’ **do**
Notice that “color” is implicitly typed with the specified discrete range by the compiler. The discrete range is ambiguous! since the compiler will not know to which type it belongs: favoriteColors or TrafficLightColors? Solution by Ada:

```ada
for color in favoriteColors ('red') .. favoriteColors ('green') do
```

C) **Subrange Types**: Pascal (and other languages like Ada) allows the programmer to define subranges of only discrete types (enumerated, int, characters), where it inherits all of its parent defined set of operations.

```ada
type uppercase = 'A'..'Z'; index = 1..100; WeekDays = Mon..Fri;
```

Advantages:
1. Enhances readability.
2. Security, assigning out of range value will be detected at compile time in case of literal constant value, or at run time in case of expression/variable assignment.
3. Efficient memory representation of the type values.

*Notice* that when the “subrange” inherits its parent set of operations, it introduces violation of the parent abstraction which leads to **security loophole**. For example, “dayOfMonth = 1..31”, it is ok to add/subtract from the day of month, but what about dividing/multiplying??!

D) **Set Type**:

Advantages:
1. High level and application oriented
2. Efficient representation.
3. ADT and readability.

```ada
type favoriteColors = (red, yellow, magenta, brown, aqua, blue, green);
setcolorset = set of favoriteColors;
```

```ada
var set1, set2 : colorset;
digits : set of char;
begin
  set1 := [red, blue, yellow, aqua];
  set2 := [brown, green];
  T := [1..10]; (* set of integers 1 to 10*)
  S := [1, 2, 5, 7, 12, 13];
  T := T * S; (* T will have 1,2,5,7 *)
  (** the operator * is used instead of the intersection ∩ **) if T = [1, 2, 5, 7] then …
digits := ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'];
read (ch);
if ch in digits the .... (* the in operation is used for membership *)

More operations on sets: \( \langle, \leq \rangle \)

**E) “Record Type”:** One of the most important contributions by PASCAL as a
“heterogeneous” data structure that aggregates a group of
related, but different data types fields that pertain to an object
(e.g., employee, student, etc). It is called “structure” in C.

**Variant Records: (pages 187-189):** Similar to the C language “union”, variant
records aim at sharing memory again, with “aliasing” will introduce a security
loophole in PASCAL!

**F) TypedList “Pointer”:** Pascal is among the pioneering languages to introduce typed
pointers (memory locations), for linked structures. C and C++
also have typed pointers. Java has no pointers, but it has
references (pointers to structures, class instances, no need to
dereferences, and it is nonsense to apply arithmetic ops on
them!).

In PASCAL: \( \text{var} \ p: ^\text{real}; \ x: \text{real}; \ c: \text{char}; \)
\begin{verbatim}
begin new (p); \hspace{1cm} \text{p}^\langle := 3.14159; \hspace{1cm} \text{c} := \text{p}^\langle ; \hspace{1cm} \{\text{compiler error!}\}
\end{verbatim}

In C&C++:
\begin{verbatim}
\text{int } *\text{ptr}; \hspace{1cm} \text{int count, init}; \hspace{1cm} *** \hspace{1cm} ****
ptr = &\text{init}; \hspace{1cm} \text{count } = *\text{ptr};
\end{verbatim}

They also have a non-types pointers “void *\text{ptr}” as generic pointers of
type any; in case we need to have a function to deal with memory blocks
of byte/words (any type). Just define the formal parameters to be “void*
\text{ptr}”, to deal with any actual parameter pointer type you send to the
function!

Questions: Is the use of non-types pointers secure? What type of tradeoff is that
of the “void *\text{ptr}” facility?

We can have pointers to any type, not only basic types (int, real, ..).
Ex: \( \text{var} \ p: ^\text{plane}; \hspace{1cm} (* \text{plane is a record type*})
****
p^.parked := ....
- A pointer with the value “nil” points to nowhere! (what is the type of a nil pointer?)

(C&C++ use value 0 instead of nil)

**No dynamic arrays, or blocks, in PASCAL!! (why?)**

No need for dynamic arrays because we have the typed pointers (efficient no range
check at run time). No need for blocks because of the associated run time overhead
of AR allocation/de-allocation, upon handling blocks’ entry and exist, respectively.
Passing Functions/Procedures as Parameters, Securely:

PASACL is the first imperative language to pass functions/procedures as parameters, “securely”.

Ex:  function test (function f (x: integer): real; x : real) real;
     begin test := f(x * x) – f(-x*x) end;

PASCAL is one of the pioneer’s languages to have the input/output statements as part of the language definition!

Writeln (“the output is →”, y);

Name Structure:

The way to bind names to their meanings in PASCAL is done via the following Six bindings mechanisms:
1- Constant  2- Type  3- Variable  4- Procedure/Function
5- Implicit Enumeration  6- Label  (yes PASCAL has “goto” a label has to be declared in the scope of its use:
  var dest : label;)

Control Structure:

The following are very important contribution of PASCAL:
The definite “for” loop:  for <name> := <expr> (to/downto) <expr> do <statement>
The indefinite “while” loop: while <condition> do <statement>
The indefinite “repeat” loop: repeat <statement> until <condition>
The “case” statement: case <expr> of <case clause>;
                       <case clause>;
                       ***
                       <case clause>
                    end
where <case clause> ::= <constant>, <constant>, ..., <constant>: <statement>

Ex:   case 1 of
      1 : begin **** end;
      2, 3 : begin **** end;
      4 : begin **** end
   end
Parameters are passed by:
1) value  2) reference ( 3- constant was used! Pp: 202-3)

The PASCAL global declaration and by-reference (or const) has the “aliasing” that might lead to insecure name access, security loophole!

------------------------------------------- 3-7-06--------------------------------------------------------

Block Structured Languages

There are two major structures that are maintained for any computation at run time:

A) Soft structure: activation state with the following two parts:

a) Fixed Part: the machine code of the program, and
b) Variable Part: The activation record (AR) that holds the state of the “activation” (procedure/function in execution).
   i) environment part (ep): defines the context of an activation; it consists of:
      a) locals and formal parameters.
      b) Static Link (SL): pointer to all visible non-local accessible scopes (starting at the AR of the definer of the activation) of the current activation.
   ii) Instruction Pointer (IP): pointer into the current activation code.
   iii) Dynamic Link (DL): a pointer to the caller’s AR, where the return address is stored.

B) Hard structure (virtual processor $\pi$): consists of the following:

   ii) Instruction Pointer (IP): a pointer to the next instruction to be executed.
   iii) Stack Pointer (SP): a pointer to the next AR record to be placed at the top of the stack, in case of the current activation calls a procedure or function.

Name Access (local/non-local):

Vocabularies:

i) Static Chain: a chain starting from the current AR’ SL until the main program AR, following the SL of all intermediate ARs.
ii) Dynamic Chain: a chain starting from the current AR’ DL until the main program AR, following the DL of all intermediate ARs.
ii) **Static Nesting Level** *(snl)*: the *snl* of a name use or declaration is the number of surrounding contour diagram boxes around its use or declaration, respectively.

The Static distance *(sd)* of a name is:

\[
\text{sd} = \text{snl}_{\text{use}} - \text{snl}_{\text{decl}}
\]

- The compiler will maintain a symbol table that keeps a record of properties (e.g., type, *snl*$_{\text{use}}$, *snl*$_{\text{def}}$, *sd*), and “offset” within the local declarations of its definer, etc) for each name; this will help in compile time type checking and run-time name access (look page 216).

**Question:** Knowing that for any used name in the program, the *snl*$_{\text{use}}$ must be greater than or equal *snl*$_{\text{def}}$ (i.e., *sd* of the name is positive), is it a sufficient condition to have positive *sd* for a name to be used? (No, the name has to also visible)

**Accessing Non-Local Names:** *(static scoping)*

In addition to the available offset in the symbol table, the compiler will compute the *sd* of every non-local name, then it generates machine code to carry out the following at run time:

a) To locate the AR of the environment of definition of the non-local, traverse the static chain *sd* times in order to get to the defining module’s AR.

b) Add the name’s offset (from the symbol table) to the obtained address of the defining AR, above (in a).

\[
\text{AP} := \text{M}[\text{EP}].\text{SL}; \text{traverse first link of the static chain to the AR of the definer of the current activation.}
\]

\[
(sd - 1) \{ \text{AP} := \text{M}[\text{AP}].\text{SL} \} ; \text{traverse remaining links of the static chain, up to the AR of the definer of the non-local.}
\]

\[
\text{fetch} \; \text{M}[\text{AP} + \text{offset}] \; ; \text{offset with the AR of the definer to get to the non-local name.}
\]

The total number of memory referencing is “sd+1”, expensive execution time in case of deeply nested modules.

**Question:** Discuss the above concern as a language designer, and as a programmer.

The language designer will investigate the language environment, if userd tend to have deeply nested modules, then think of other mechanism not the “static chain”. If the programmers are using a language with “static chain”, then they should not deeply nest proc/functions, for their program to execute faster!
Question: What is the scenario that makes the static and dynamic chains the same?
When the callers and the definers are the same.

Question: Show a scenario where the static and dynamic chains are not the same.

---

Procedure “Call” Sequence with Static Chain:

\[
\begin{align*}
M[\pi_{SP}]\text{.PAR}[1] & := \text{evaluate } (\text{actual-parameter}[1]); \\
M[\pi_{SP}]\text{.PAR}[2] & := \text{evaluate } (\text{actual-parameter}[2]); \\
&& \text{***} \\
M[\pi_{SP}]\text{.PAR}[n] & := \text{evaluate } (\text{actual-parameter}[n]); \\
M[\pi_{EP}]\text{.IP} & := \pi_{IP} \quad \text{; save resume address into the caller’s AR} \\
M[\pi_{SP}]\text{.DL} & := \pi_{EP} \quad \text{; set the dynamic link in the callee’s AR} \\
sd * (\pi_{EP} & := M[\pi_{EP}]\text{.SL}) \quad \text{; get to the environment of the callee’s definer} \\
M[\pi_{SP}]\text{.SL} & := \pi_{EP} \quad \text{; set the static link of the callee} \\
\pi_{EP} & := \pi_{SP} \quad \text{; set the virtual processor EP to the callee AR} \\
\pi_{SP} & := \pi_{SP} + \text{size (callee’s AR)} \quad \text{; set the hard virtual processor stack pointer ready for the next AR allocation for any future proc/function call!} \\
\pi_{IP} & := \text{entry address of the callee’s code} \quad \text{; transfer the control to execute at the callee’s code by the proper setting of the virtual processor’s IP.}
\end{align*}
\]
Procedure “Return” Sequence with Static Chain:

1) $\pi_{SP} := \pi_{SP} - \text{size}$ (callee’s AR) ; pop the callee’s AR from the stack

2) $\pi_{EP} := M[\pi_{EP}].DL$ ; reactivate the virtual processor back into the caller

3) $\pi_{IP} := M[\pi_{EP}].IP$ ; goto the resume address and start executing at the caller’s code

Question: Why do not we store the return address at the callee side, instead of the caller side?

In the “return” sequence above, we can not de-allocate the callee’s AR via the decrement of the $\pi_{SP}$ (1) at the beginning, since after we update the $\pi_{SP}$ (1), we can not use it to access the callee’s AR anymore (to get the return address and the dynamic link). Hence, we have to move (1) at the end, instead of (3). But, once we execute (3) updating the $\pi_{IP}$ (2), the execution control will transfer to the caller’s code, hence the statement to adjust the $\pi_{SP}$ (at its new order) will never be executed.