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 governed 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 evaluates 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”.

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

A) HLLs Translators:

i) **Compilers:**

HLL program → **Scanner** → Token stream → **Syntactic Analysis** →
Abstract syntax parse tree → **Semantic Analysis** → Abstract syntax parse tree
→ **Optimization** → **Code Generation** → 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 → Assembler → executable machine code

B) Software simulators (Interpreters):

HLL program → **The HLL’s Virtual Machine** → 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 = Y/Z + F(4)
     ii) Control Flow:   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:   GO TO label
  ii) Computed:    GO TO (L1, L2, …, L n), I
      If I = k, jump to Lk label in the label list, 1 <= k <= n; otherwise no jump.
  iii) Assigned:   GO TO  N, (L1, L2, …, L n)           [ 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  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.

\[
X + I \rightarrow X + \text{FLOAT}(I) \quad X = I \rightarrow X = \text{FLOAT}(I) \quad I = X \rightarrow I = \text{IFIX}(X)
\]

- **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., \( N = \text{ISUCC}(6\text{HCARAMEL}) \).

  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
  
  CALL ALTERNUMERIC(7)
  I= 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:

```
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:

```
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, …, Pn) →

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[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., \( I = \text{IFIX} (X + \text{FLOAT} (I)) \) 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, \( X^{\frac{1}{3}} \) 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:

\[
\begin{align*}
\text{SUBROUTINE SUB1( A, B) } & \quad \text{SUBROUTINE SUB2( H, M) } \\
\text{REAL X(100) } & \quad \text{REAL C(50), D(100) } \\
\text{INTEGER Y(250) } & \quad \text{INTEGER E(200) } \\
\text{COMMON /BLOCK1/ X, Y } & \quad \text{COMMON /BLOCK1/ C, D, E } \\
\text{END } & \quad \text{END }
\end{align*}
\]
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)\textsuperscript{th} element of the array IF will be assigned 123.

\begin{align*}
\textbf{ALGOL60 (Naur, 1960)}
\end{align*}

- Elegancy and generality (powerful universal ALGOrithmic 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: \texttt{begin} declaration-sequence; \texttt{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.

\[
x := 0;\ y := 1;
\text{for } i := 1 \text{ to } 2 \text{ do}
x := x + i;
y := y \times x;
\]

(* 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?

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/EQUIVALENCE, 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: \* for i := 1 step 2 until N * M do \* statement;
   - Indefinite looping: for NewGuess := Improve (OldGuess)
     \* while abs(NewGuess – OldGuess) > 0.0001 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)

   ```
   for i := 3, 7,
   11 step 1 until 16,
   i/2 while i ≥ 1 ,
   2 step i until 32
   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”:
  ```
  begin
  switch S = L , if i > 0 then M else N , Q ;
  (evaluate i, j)
  goto S[j]; (* if j=2 then the value of i will decide jumping to either M or N *)
  ```
  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:
\[
<expr> ::= <expr> + <term> \\
| <expr> - <term> \\
| <term>
\]
\[
<term> ::= <term> * <factor> \\
| <term> / <factor> \\
| <factor>
\]
\[
<factor> ::= <id> | <number> | (<expr>)
\]

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: \( <if-stmt> ::= if (<logic-expr>) then <stmt> [else <stmt>] ; \)
2- zero or more repetitions of some constructs using braces.
Ex: \( <stmt-list> ::= <stmt> {; <stmt>} \)
3- multiple-choice options:
Ex: \( <for-stmt> ::= for <id> := <expr> (to|downto) <expr> do <stmt> \)

Attribute Grammar to Describe Context Sensitive Languages Aspects:

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

***
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:
```pascal
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.

```pascal
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.

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

```pascal
var set1, set2 : colorset;
digits : set of char;
```

```pascal
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: $\langle, \leq, \geq$

E) “Record Type”: One of the most important contribution 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) Typed “Pointer”: Pascal is among the pioneer 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: var p: ^ real; x: real; c : char;
begin new (p); p^ := 3.14159; c := p^ ; {compiler error!}

In C&C++:
int *ptr; int count, init; *** ****

ptr = &init; count = *ptr;
They also have a non-types pointers “void *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 *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 *ptr” facility?
We can have pointers to any type, not only basic types (int, real, ..).
Ex: var p: ^ plane; (* 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:  

```pascal
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!

```pascal
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: 
```pascal
var dest : label;
```

Control Structure:

The following are very important contribution of PASCAL:

The definite “for” loop:  
```pascal
for <name> := <expr> (to/downto) <expr> do <statement>
```

The indefinite “while” loop:  
```pascal
while <condition> do <statement>
```

The indefinite “repeat” loop:  
```pascal
repeat <statement> until <condition>
```

The “case” statement:  
```pascal
case <expr> of
<case clause>;
<case clause>;
***
<case clause>
end
```

where <case clause> ::= <constant>, <constant>, ..., <constant>: <statement>

Ex:  
```pascal
case l 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:

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

- The compiler will maintain a symbol table that keeps a record of properties (e.g., type, \text{snl}_{\text{use}}, \text{snl}_{\text{def}}, \text{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 \text{snl}_{\text{use}} must be greater than or equal \text{snl}_{\text{def}} (i.e., \text{sd} of the name is positive), is it a sufficient condition to have positive \text{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 \text{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).

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

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

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

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 users tend to have deeply nested modules, and 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.

----------------------------------- 3-9-06 --------------------------------------------

**Procedure “Call” Sequence with Static Chain:**

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

\text{resume:} \quad (\text{house keeping code}) \ ; \text{return address}
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.

Procedural Parameters Are Represented by “Closures”:

- We know that the state of an integer or real name is its value (e.g., 100, 2.2876, respectively), but for the type “procedure/function” the question is: what is the state (value) of such type, and how to represent it?
- The answer is the “closure”: A “closure” is $<ip, ep>$ where $ip$ is the code instruction pointer; and the $ep$ is a pointer to the environment of the definition of the procedure/function that is represented with such closure.
- The compiler, will evaluate a closure for every actual procedure parameter; and passes it to its corresponding formal. Then, inside the callee, if such formal parameter procedure (function) is called, the compiler will generate the following machine code calling sequence.
Calling Formal Parameter Procedure ($f_p$)

$$M[\pi_{SP}].PAR[1] := \text{evaluate} \ (\text{actual-parameter}[1]);$$
$$M[\pi_{SP}].PAR[2] := \text{evaluate} \ (\text{actual-parameter}[2]);$$

***

$$M[\pi_{SP}].PAR[n] := \text{evaluate} \ (\text{actual-parameter}[n]);$$
$$M[\pi_{EP}].IP := \pi_{IP} ; \text{save resume address into the caller’s AR}$$
$$M[\pi_{SP}].DL := \pi_{EP} ; \text{set the dynamic link in the callee’s AR}$$

$$M[\pi_{SP}].SL := \text{fp.ep} ; \text{fill in the SL of fp’s AR, with the environment}$$
$$\pi_{EP} := \pi_{SP} \ ; \text{set the virtual processor EP to the callee AR}$$
$$\pi_{SP} := \pi_{SP} + \text{size (callee’s AR)} ; \text{set the hard virtual processor stack}$$
$$\pi_{IP} := \text{fp.ip} ; \text{go to the code of fp}$$
$$\pi_{IP} := \text{fp.ip} ; \text{go to the code of fp}$$

* Notice that we assumed that the fp’s closure exists and we used it, but at
some point the compiler has to build it as follows:

**How to Build the Closure (for procedure $p$)?**

$$M[\pi_{SP}].PAR[i].ip := \text{entry} \ (p) ; \text{build the ip part of the closure of procedure } p$$
$$\pi_{AP} := M[\pi_{EP}].SL \ ; \text{traverse the first static link}$$
$$(sd-1) \ (\pi_{AP} := M[\pi_{AP}].SL) \ ; \text{get to the environment of the p’s definition}$$
$$M[\pi_{SP}].PAR[i].ep := \pi_{AP} ; \text{build the ep part of the closure of procedure } p$$

Look page 226, figure 6.5 for examples of formal procedure parameters.
Functions/procedures As First Class Citizens (FCC)

- Until now we passed functions/procedures as parameters; what about returning them as values from other functions (i.e., treating them as FCC)?
- Function Composition:

1. Program Test;
2. type
3. fun : function (integer): integer;
4. var
5. m: integer;
6. h, incr, sqr : fun;

7. function Compose (f, g : fun): fun;

8. function apply_comp (x : integer): int;
9. begin
10. return (f (g (x) ) );
11. end;

12. begin (* Compose *)
13. return (apply_comp);
14. end (* Compose *);

15. begin (* Test*)
16. h := Compose (incr, sqr);
17. m := h (3)  (* m = 10 *)
18. end. (* Test*)

If we follow the run time stack dynamic formation, we will find out that upon exiting from Compose line 16, there will be a closure <ip, ep> apply_comp With its ep pointing to its definer “Compose”, but we already exited from Compose! Hence, there is a problem of an orphan “ep”!!!!!!!!!!!!!!!!!!!!!
To solve the above problem, we must give up the “stack” as a model of computation, and use a “heap”, where we can keep an activation record even after we exit its module!

Thus, for languages that return functions/procedures from other functions, we can not use the “stack” as a model of computation, at run time; instead we use a “heap”.

**Retention:**
It is the ability to retain the AR of a procedure/function, even after exiting its code. Such retention can not be achieved in languages that utilize the “stack” as their model of computation. It is used in languages that uses “heap” as their model of computation.

Question: Would the feature of “retention” violate the static scoping rules manifested by the contour diagram semantics????

Answer: Yes! A deeply nested function can be returned to a higher snl, exposing it to be called by a lesser snl procedure!

-----------------------------------------------------------------------------------------------

**MIDTERM EXAM**

-----------------------------------------------------------------------------------

**MODULA-2**

- The design is centered around the modularity abstraction of the “module” unit, which aids in the design of reliable, cost-effective, large, and complex software systems. The modular facility provides for software partitioning into logical units, each with well defined interface that is separate from the implementation part.

- The language combines some low level facilities, in addition to the high level modular abstraction.

- Very PASCAL like (after all, the same designer, Wirth 1980), but with the followings:

  1) No “go to”! and the language is case sensitive.
2) Only 40 reserved words, but there are 30 user predefined “standard identifiers” (Yakii!!): INTEGER, REAL, BOOLEAN, CHAR, TRUE, FALSE, NIL, ABS, MAX, MIN,…

3) No need for the statement grouping via the compound statements, every structuring statement has its own delimiters, e.g., IF …ELSEIF… END, FOR … END.

4) The declaration of type “PROC” only for system procedures, where we can declare variables of type “PROC” and assign to it “sin” “cos” system functions. First try into treating functions as “first class citizens”

5) Low level facilities, system calls: “WAIT” and “SIGNAL” that are used for mutual exclusion facilities to guard “critical” sections of codes that are shared among “concurrent” processes.

6) New types: “PROC”, LONGINT, CARDINAL (unsigned int), LONGREAL, BITSET, WORD (matching “any” type of word size), ARRAY OF WORD (for structure “any” multiple word size), ADDRESS (actual memory address, un-typed) which is compatible with any pointer type. Also, low level functions such as: TSIZE(type) that return the size of the type, at different hardwares, HIGH(item) that returns how many words in the item. In addition, it defines “literal constants”: OCTAL- 177B, Hex- 7F99H, Char-101C (=A), 7C (Bell). “BITSET” built in set type:

```pascal
VAR A: ARRAY [1..6] of BITSET;
BEGIN   A[1] := {0-7,9,15} ;
    (* the i^th array cell has 1 in positions: 0 to 7, 9,and 15*)
```

7) New control structures:
IF … THEN … ELSEIF … END
Looping: infinite loop,

```
LOOP: -------
   IF cond THEN EXIT
-------
   IF cond THEN EXIT
-------
END
```
8) The support of “true” ADTs via the modular facility, four types of modules, with the IMPORT/EXPORT mechanisms.
9) The internal module as static scope definer (has a much more controlled name scoping than ALGOL blocks).
10) Input/Output commands are not part of the language.
11) No built in file types (Input/Output file system) as in PASCAL.

-------------------------------------- 3-31-06-------------------------------------------

**Modules**

There are four types of modules in Modula-2:

i) **Program module**: the main program and it does not export any names. It may contain internal modules or procedures and import from lib modules. (e.g., the “stacktest” module in your hand out of the stack ADT)

ii) **Internal Modules**: mainly for controlled name scoping in and out of its contour via the IMPORT/EXPORT, an evolution from the ALGOL blocks.

Internal Modules non-local (external) name visibility rules:
Local names (local declarations or exported by hosted modules, children) are seen automatically.
But, for non-locals (externals) the story is different.

```
MODULE N;
IMPORT z; (* assume z is visible outside N*)
VAR a, b: REAL;
MODULE M;
IMPORT a, z, h; EXPORT d;
VAR c, d: CHAR;

MODULE L; (* child hosted by M*)
VAR x, y: REAL; EXPORT x;
BEGIN (*L*)
(* visible names: local declarations → x, y *)
END L;
```
BEGIN (* M*)
(* visible names in M: 
local declarations → c, d (internal); 
and 
EXPORTED by child L → x (internal) 
and 
IMPORTED, visible external,
by N → a (local), z (imported)
by M1 → h (exported) *)
END M;

MODULE M1; (* M1 is a sibling of M *)
EXPORT h;
VAR f, h: CHAR;
BEGIN (* M1 *)
(* visible names: local declarations → f, h *)
END M1;

BEGIN (* N *)
(* visible names in N: 
local declarations → a, b; 
and 
EXPORTED (internal) by M → d, and 
by M1 → h ; 
and 
IMPORTED by N → z (visible external) *)
END N;

a) An internal module “M” can use, in its code, an external name only if the following two conditions hold:
1) the name is visible to the contour (module/procedure) immediately hosting M (i.e., N) as locally declared, visible imported name, or exported by a sibling of M (i.e., M1) within N, and
2) M imports such visible name.
b) An internal module “M” can use, in its code, an *internal* name only if it is locally declared or exported only by all immediately nested modular contours (e.g., L).

iii) Definition Module: It is the interface (SPECs) of the lib module which lists all exported/imported names and proc/func headers from/to the lib module, respectively.

iv) Implementation Module: It has the concrete “coding” implementation of all exported names and proc/func headers (types and operators). It is body might have trivial code!

* Library modules are compiled separately, allowing for fast software development. The user of an imported lib module needs only its compiled interface, in order compile the user code.
* A type name that is exported in a module definition without any info about its concrete implementation is called “opaque” type. In such case the concrete implementation of its operations must be written in its exporting implementation module, where its concrete implementation is listed.
* Module introduced the power of genericity (polymorphism) via the *generic* (open) types: “WORD” and “ARRAY OF WORD”. Yet, since same size does not mean same logical type, the open types in Modula-2 introduces a security loophole. The reason is Modula-2 is statically typed checked; hence the compiler MUST carry out its type checking at compile time. There is no way to do so, since an open type formal parameter is compatible many logical types (as long as they are of the same size), thus the compiler can not carry any type checking that involves a formal open type parameter! It is obvious that this is a case of “power” versus “security”. We gain power with type any but we loose security because of the (efficient, yet) static type checking. If the designer of Modula-2 selected the dynamic type checking as a solution, it would work, BUT they would have lost in the direction of efficiency (why?).
Ada (DoD, 1983)

- Imperative, Modula-2 & Pascal like for programming “in-large” and “real-time” “secure” embedded military systems.
  ** Parallel tasks (rendezvous).
  ** Robustness (exception mechanism) & reliability.
  ** Modularity & Info hiding & portability (package facility)
  ** Polymorphism (static): generic packages; overloading.
  ** Very rich (the richest) & strong typing system.

* Back to Algol’s “blocks”:
  
  `<name>`: declare <decl-seq>
  begin <stmt-seq>
      [exception <exp-handler>]
  end [<name>]

* Ada is case insensitive (FOO=foo).

* Ada types as in Pascal & Modula-2, plus:

  i) **Typed & untyped constant declarations:**

    A: `constant FLOAT := 1000.00 ;`
    B: `constant INTEGER := f(A);`  -- f is a function that returns integer
    E_C `constant := 0.57721;`

    Light_Year := `5_878_000_000_000 ;`  -- “named number” for readability

    There is no explicit type of the above “named number”, instead the compiler will assign it an internal “universal” (LONG_INTEGER) anonymous type. The reason is that to leave it for the compiler to decide, based on the underlying hardware, the suitable representation; hence make programs portable.

  ii) **Real types**: the Ada compiler provides FLOAT, LONG_FLOAT, SHORT_FLOAT; implementation dependent and not very portable.

  **For portability:**
  type DISTANCE is digit 6  -- the programmer specifies a 6 digits of accuracy in the “mantissa”. If the hardware can not provide, the compiler will reject; otherwise it will select one of the above REAL types.
iii) **Subtypes:**

```plaintext
type DAY is (MON, TUE, WED, THUR, FRI, SAT, SUN);

subtype BANKING_DAY is DAY range MON..FRI;
subtype WEEK_END is DAY range SAT..SUN;
subtype SHORT_TERM is INTEGER range 1..365;
subtype ACCOUNT_BALANCE is DOLLAR range 100.00..100_000.00;
subtype ACCOUNT_ID is POSITIVE range 1..1e9;
```

S : SHORT_TERM; J: INTEGER;

J := S; -- OK! (why?)
J := S; -- Passes compile time, but forces a run time “constraint” check
       -- via compiler generated machine code.
S := 0; -- Legal at compile time, but causes a CONSTRAINT_ERROR at
       -- run time.

Remember that the compiler does not concern itself (for logical and design
aspects) at all with the value calculation of any entity (name or literal),
instead it focuses only on the “type” of such entity, leaving the value
decisions to the CPU at run time (yet, it generates and deposits the code
that carries out such calculation). For example, “S := 0” is very clear error,
for us only, not the compiler since we did the a-to-b conversion, and not
the compiler!

Examples of using subtypes:
TODAY : DAY;

**If** TODAY in WEEK_END **then** SLEEP_LATE;
**for** I in BANKING_DAY **loop** … **end loop;**
**case** TODAY is
  **when** BANKING_DAY  ……;
  **when** WEEK_END  ……………;
  **others**  null ; -- YES there is “null” statement in Ada
**end case;**

**Attributes:** They are special operations to obtain properties of types,
objects, and subtypes, at run-time.
DAY’FIRST → MON
DAY’LAST → SUN
I := 101;
INTEGER’IMAGE (I) → “101” string (b-to-a)
INTEGER’VALUE (“101”) → 101 integer value (a-to-b)

subtype INDEX_RANGE is 1..200;

for I in INDEX_RANGE’RANGE loop ..... end loop;

iv) ARRAYS:

type FLOAT_VEC is array (1..N) of FLOAT;

Ada has dynamic arrays (as shown above, where arrays size, N, need not to be known at compile time), N has to be declared and has value before the elaboration (expansion of declarations at run time, including memory allocation) of names of type “FLOAT_VEC”.

N : INTEGER := f(x); -- run time evaluation of the value of N, calling f
A, B: FLOAT_VEC; -- elaboration of the FLOAT_VEC is done at run-
-- time, as an array of 400 cells of FLOAT.

Operations on arrays:

A := (1.0, 0.0, 0.0) -- array of three cells initialized with the given values.
A:= (1.0, othersÆ0.0); -- A has the first cell = 1.0, and the rest (399) of
the cells have 0.0

A(1..31) := B(335..365); -- both, A and B must be of the same type.

Can we do the above array slice assignment in case of the following A&B declarations?

A: array (1..400) of INTEGER; B: array (1..400) of INTEGER;

The answer is NO., because in the above declarations, the compiler will assign two (different) anonymous internal type names for A and B; hence they are not the “same” type!
Question: Why do you think ALGOL has dynamic arrays, but PASCAL did not have them? ALGOL did have pointer for dynamic structures, whereas PASCAL has pointers.

Question: Why did Ada include dynamic arrays, even with the inclusion of pointers? Ada gives the user the choice of using either (DAs or PTRs) based on the application in-hand, if the dynamic array is not involved in many instruction, i.e., not generating too many runtime “index constraint checks”, then DAs are used, otherwise pointers are used.

Unconstrained Arrays: The array range is not written in the type declaration (considered not part of the type, by Ada), instead it is given when the type is to be bound to a name (name declaration). In PASCAL, the range is part of the type, two arrays of different ranges have different types, and thus it does not have the unconstraint facility!

type UCA is array (INTEGER range<>) of INTEGER;

A: UCA(1..365); -- static instantiation of the type range
B: UCA(200..400); -- static instantiation of the type range

Both arrays have the same type, however with different index constraints. In addition to the ability to write polymorphic “type” declaration (UCA), they facilitate the construction of polymorphic procedures and functions.

procedure SUM_ARRAY (X: in UCA; S: out INTEGER) is
  begin
    S: INTEGER := 0;
    for I in X’RANGE loop S:=S+X(I); end loop;
  end SUM_ARRAY;

The attribute “RANGE”, above, is a shorthand for the range “X’FIRST..X’LAST”.

type STRING is array (POSITIVE range<>) of CHARACTER;

subtype PASSWORD_TYPE is STRING(1..8);
subtype CODE is STRING(1..32);
subtype LINE is STRING(1..80);
We can not use an unconstrained array as a base type for other arrays, arrays base types must be constrained.

```ada
  type PAGE is array (1..500) of STRING;  -- illegal
```

**But**

```ada
  type PAGE is array (1..500) of STRING(1..80);   -- legal
  type PAGE is array (1..500) of LINE;
```

------------------------------------------------------ ---- 4-6-06 ------------------------------

v) **RECORDS:** A named aggregate (composite) type to group related set of names (fields).

```ada
  type TRANSACTION is record
      ACCOUNT     : ACCOUNT_ID;         -- see section iii above
      PASSWORD  : PASSWORD_TYPE  -- see section iv above
      BALANCE     : REAL   := 0.0;
    end record;

  THIS_TRANSACTION : TRANSACTION;
```

Records may be assigned values as whole (like arrays), via:

Positional association:

```ada
  THIS_TRANSACTION := (55_238_1245, “TOUGH2GET”, 125.00); or
```

Named association:

```ada
  THIS_TRANSACTION := (ACCOUNT  55_238_1245,
                     PASSWORD  “TOUGH2GET”,
                     BALANCE  125.00);  -- more readable
```

**Discriminant Records:**

What about if we need to accommodate different types of records that have different lengths passwords. Instead of declaring separate record type for each, Ada allows the use of a **discriminant** to parameterize the record for any varying field value, while keeping the same record type.
type TRANSACTION (PASSWORD_SIZE : POSITIVE) is record
    ACCOUNT : ACCOUNT_ID; -- see section iii above
    PASSWORD : STRING (1..PASSWORD_SIZE);
    BALANCE   : REAL := 0.0;
end record;

the declaration of different sizes passwords records, based on the parameterized discriminant:

A1, A2 : TRANSACTION(3); -- PSWD of 3 chars
B     : (PASSWORD_SIZE \to N);
-- B has N chars password; if N is variable, then its value is not
known until run time when elaborating the declaration!
(dynamic “password” array)
A1 := A2; --ok
A1 := (3, 55_238_1245, “4u2”, 125.00); -- ok
A1 := B -- ok if N is 3 at run time!

Variant Record Facility:
It is used to declare one record type with variant info for some of its fields, that is used to declare records that varies in some field.

type PET is (DOG, CAT, PARROT, SNAKE, RABBIT);
type HOUSE_PET (KIND : PET) is record
    AGE : NATURAL; -- non-zero positive integer
    case KIND is
        when DOG \to
            HOUSEBROKEN : BOOLEAN;
        when CAT \to
            FUSSY : BOOLEAN;
        when PARROT \to
            VOCABULARY : NATURAL;
        when others \to -- it covers for the rest of pets, and must
            -- be there!
            null; -- no component exists for such variant!
    end case;
end record;
Declarations and name binding to type and memory allocated spaces of five PET records, with different third filed type, based on the discriminant instantiation provided values:

GIANT : HOUSE_PET(DOG) := (DOG, 5, FALSE);
PIERRE : HOUSE_PET(CAT) := (CAT, 1, TRUE);
TALKATIVE : HOUSE_PET(PARROT) := (PARROT, 1, 100);
POLLY : HOUSE_PET(DOG) := (DOG, 5, FALSE);
POWER : HOUSE_PET(SNAKE) := (SNAKE, 2); -- only two fields!

Remember, the above declarations are all *elaborated* at run-time, thus any misuse (see below) will not be notices at compile time, instead it will cause “constraint-error” at run time

GAIN.T.HOUSEBROKEN := TRUE; -- OK
GAIN.T.VOCABULARY := 5; -- will cause CONSTRAINT_ERROR

Question: Are variant records still a security loophole in Ada?

NO, read top of page 251. Insecurity happens if we change the discriminant (tag) of a variant record without reinitializing the entire record! For example: GAIN.T.KIND := PARROT; -- is illegal.
Otherwise, whatever leftover value from the “DOG” GAIN in its old “BOOLEAN” filed “HOUSEBROKEN” would be used as the amount of “VOCABULARY” to its new state as a PARROT!
Hence, Ada enforces the rule: you can not single out the discriminant filed only in record and assign it a new value, you must assign a complete record to the record in hand.

vi) *Access Types (Pointers)*

type TRANSACTION_PTR is access TRANSACTION;

FIRST_TRANSACTION, SECOND_TRANSACTION:

TRANSACTION_PTR; -- declare 2 ptrs to trans.

FIRST_TRANSACTION :=

new TRANSACTION_PTR; -- return ptr to first trans
FIRST_TRANSACTION.AMOUNT := 25.00; -- initialize its fields
FIRST_TRANSACTION.ACCOUNT := 55_238_1234;
FIRST TRANSACTION.PASSWORD := “UPDOWN”;
Notice there is no use of dereferencing operator (as in Pascal and C); it is implicit in Ada (just use the ‘.’ projection notation)

Assign the fields’ values of an already existing record to a dynamically allocated record:
SECOND TRANSACTION.all := FIRST TRANSACTION; -- No aliasing

Assignment of two dynamically allocated records, T1 and T2:
T1.all := T2.all ;

**Aliasing** two pointers to the same record:
T1 := T2; -- pointer assignment only, no dynamic allocation of T1.

A default value of a pointer is “null”.

vii) **Derived Types:** They allow the definitions of new types driven from existing types. Even though the derived types inherit the parent type’s operations, they are not compatible (logically distinctive)! To make them compatible we need to use explicit conversion.

```
type PERCEN is new INTEGER range 0..100;
P : PERCEN; I: INTEGER;
P:= I; -- illegal also I := P; -- illegal
```

But:  P := PERCEN(I); -- is legal
I:= INTEGER(P); -- is legal
PACKAGES

“Packaging” is the Ada basic data abstraction and modularity facility to group logically related data and their associated set of operations, forming an environment that can be accessed via mutual consent between the user and the implementer of the package.

i) Package Interface (Specification):

package <NAME> is
   -- visible section: objects and type declarations; procedure/function headers

   [ -- private section: optional section that has private types and all deferred constructs concrete declarations. It is to be seen by the compiler not by the users, for optimal space allocation by the compiler.]

end [(optional) <NAME>]

ii) Package Body (Implementation):

package body <NAME> is

   [ -- optional: 1) local declarations, invisible to the users
      2) concrete implementation of the procedures and functions
         package operations (above). ]

   [--Optional section that with some Ada statements, e.g., for initialization ]
   [exception -- Optional section that with Ada exception handlers ]

end [(optional) <NAME>];

Package’s “Specification” and “Body” can be compiled separately (same as in Modula-2). A package can be lib module, or nested in subprograms.
• Packages can be a lib modules and separate compilation units.
• “privat” types are similar to the Modula-2 “opaque” types, since the user can not access them.
• There is no explicit “import/export” commands, yet the idea of mutual consent (user/implementer) visibility still holds, all names defined at the package interface and outside the private section are visible to the package’s users (i.e., public names).
• To invoke a lib package we use the “with” clause, in the declaration of other subprograms:
  
  with <package-name> ;
  and we utilize the “use” clause to make the package name visible in the code with no need for prefixing all of its operations with its name via the dot notation, i.e., <package-name>.<operation-name>.

  use <package-name>;
Notice: the with must be before the use!

**Generic Packaging Facility:**

• A second order static polymorphism (power) to textually group a number of different versions of an ADT (e.g., based on elements’ types or numbers) into one “generic” parameterized template to be instantiated at compile yielding concrete package definitions based on the provided actual values of the generic parameters.

**generic** -- (an example at the bottom of page 273, generic package “stack”)
  <list of dummy formal arguments>

**package** <NAME> is
  <proc/func’s headers (ops) containing some of the above arguments>
end <NAME>;

**package body** <NAME> is -- (an example at the top of page 275, generic package “stack”)
  <the concrete implementation of all names and operations that contains all of the above arguments>
end <NAME>;

• The instantiation process is simple a textual replacement of every instantiation statement with a copy of the generic package template after
replacing every dummy generic parameter by its corresponding actual argument that has been provided in the instantiation statement.

```ada
package <INSTANT-PACKAGE-NAME> is new <GENERIC-PACKAGE-NAME> (list of actual arguments values);
--(for example: look near the top of page 274)
```

- It is completely carried out by the compiler, thus it’s a “static” genericity of limited power since each instantiated package will use a separate set of operations’ codes; no code sharing.
- The goal is to shorten the Ada programs helping in the code maintenance, and program coding. BUT, we will have a problem. A one page Ada code (filled with many generic instantiation statements) might “explode” into 100 pages of text (code explosion problem in Ada), with huge compiled machine code due to the non-sharing of the operations between different instances with different type elements.
- A truly powerful instantiation would allow all versions to share the same codes of operations, which is not successfully done in Ada.
- Attempts to share operations codes in Ada have been failed due to the same back draw and insecure assumption that “same size” means “same type”; the same failure at Modula-2 side.

**Internal Versus External ADT Implementation in Ada:**

**Internal Representation of types:**

- Internal representation of an ADT resembles the object orientation view of ADTs (in OOLs). As shown in the “Stack1” example in pages 268-270, we find that the type part of the ADT (i.e., “ST” type) is listed in the package interface, thus it not exported for use outside the package. Also, its concrete implementation is done (“internally”) inside the body implementation (top of page 270). In the above internal representation of the stack ADT, we deal with the stack as an integrated “object”, not its type (“ST”), sending it messages and expecting that it will act on itself (how? we do not care). For example, at mid of page 269:
  ```ada
  declare
  use Stack1; -- assuming it is visible and not a lib package
  ....
  Push(I); -- a message sent to Stack1 ADT to push I in itself.
  Pop(N); -- a message sent to Stack1 ADT to push I in itself.
  ```
In order to declare another stack, we need to write another package (inefficient!!), or use the generic package facility, ending up with duplicate set of the same operations set, one per every stack. The problem is that even though we view the stack as an OO object, it is not a “class”!

The type “\texttt{ST}” is truly “\texttt{opaque}”, since its name and implementation are hidden inside the body, not listed at the interface for exportation.

\textbf{External Representation (Value) of types:}

For more efficient packaging of ADTs in Ada, we can use the “value” approach of representing the ADT types. In this case we externally export the “type” as a value to the users, to be used to declare more than one ADTs that share the “same” set of operations in the original package. In the example of the “\texttt{Stack\_Type}” package, page 276-277, we export the type part of the ADT (“\texttt{Stack}”) by listing its name in the package interface, and we implement it in the “\texttt{private}” section, for compilation efficiency; though still an “\texttt{opaque}” type (why?).

At the user code, we can instantiate many versions of the same stack by simply declaring them with the same exported type (‘\texttt{Stack}’):

\begin{verbatim}
declare
  use Stack\_Type;  -- assume it is not a lib package
  Stack1: Stack := new Stack;
  Stack2: Stack := new Stack;
      *  -- instantiate 100 stacks that share the same set of operations, at declaration “elaboration” (run) time
      *
    Stack100: Stack := new Stack;

begin
  Push (Stack87, I)  -- in the instantiated stack87, push I.

  We are not sending a message to anyone, instead we are invoking the “Push” operation (of the type Stack\_Type) passing it our stack as an actual parameter value (the corresponding formal is in-out) to be modified by pushing “I”.

Questions: Is the above implementation considered \texttt{polymorphic}? Justify.
Yes, since more than one instantiated stacks are aliasing on the same set of operations.
\end{verbatim}
If the answer is yes, is it “dynamic” or “static”? Dynamic, since it is carried out at the declaration elaboration time.

Is it “genericity”? NO, since all stacks are of the “same” type, had the same set of operations been used to handle different type elements stacks, then we would have called it “dynamic” “genericity”.

Parameter Passing In Ada

Ada solved the problematic of the tangled user and compiler views of parameters by making them “orthogonal” (independent).

The user informs the compiler of his/her view of the formal parameter:

i) “in”: used to input a value into the subprogram, and nothing else. Thus, the compiler will issue an error when the user attempt to change the value of such formal parameter, e.g., when used as a left hand side of an assignment. The parameter acts as constant.

ii) “out”: used to output a value from the subprogram. Old versions of Ada disallowed its use in the right hand side of an assignment, or as part of an expression. But, Ada 95 allowed that based on its implementation as the “result” half of by “value-result” in FORTRAN! Hence, Ada 95 allows the utilization of its value, assuming its been modified before use (??!!), even when the compiler implement it by reference since there will not be any intermediate change to its corresponding actual, until the return from the subprogram.

iii) “in out”: used to input a value to a subprogram, and also output a value from the subprogram.

The compiler implements the parameter passing based on efficiency, regardless of the user view. If the actual is large size composite data structure (e.g., big array) the compiler will pass a reference to the actual parameter into its corresponding formal, otherwise (i.e., scalar actual) it passes its value.

Question: Do we have a problem in Ada 95 implementation of the “out” parameter passing? YES! (explain).
The Tasking Facility In Ada

It is used to allow concurrent tasking, where tasks can be of non-communicating and communicating natures:

A) Non-communicating Tasks:

```
procedure P is
  task  T_1;  ... (SPECS interface of task T_1)... ; end T_1;
  task body T_1 is
    begin ... end T_1;

  ***

  task  T_n;  ... (SPECS interface of task T_n)... ; end T_n;
  task body T_n is
    begin ... end T_n;

begin -- code for P
  tasks T_1 ... T_n will start once we enter the code of P
  without any explicit call statement for each.
  P will not terminate until all of its tasks terminate!
end P;
```

Tasks can not be a lib unit, they must be declared within some program unit. Ada will prevent any subprogram that hosts any task(s) from terminating until all of its hosted tasks terminate. The reason is related to the disappearance (upon return/termination) of an environment (that of the hosting subprogram) that is needed by the task(s), otherwise they become orphans.

B) Communicating Tasks Communicate and Synchronize via “Rendezvous”:

Two tasks (client and server) will rendezvous via an “entry” (procedure like construct) which is to be declared in the server and “called” by the client upon requesting some service (from the server). The client “call” on some server “entry” must be “accepted” by the server, pending on the server availability, before any service is granted.
At the client: the format of an entry call is \(<server-name>.<entry-name>\).

At the server side there will be an “accept” statement that protect some “critical” code, which manipulate a shared memory space, for mutual exclusion access among the communicating tasks that share such common memory space.

\[
\text{accept } <\text{entry-name}> \text{ do }
\text{ <body, critical code> }
\text{end;}
\]

In case of more than one client tasks asking for the same entry service, all requests are to be queued (FCFS), whenever the service is not ready (busy at other entries), until the server is ready (accepting from such entry) to accept requests for that entry again, then it picks the from the head of the waiting queue of requests. Hence, the entry “call” is like a \textit{message} and the “entry” itself is like a \textit{mailbox}.

\textbf{The Rendezvous protocol:}

1) If the client issues a call (message) before the server is ready to accept, then the client task will be blocked (forced to wait) until its call (request) is accepted, leaving its call message in the server entry mailbox.

2) If the server reaches an accept statement (mail box) and find no client waiting at it (i.e., the mailbox is empty), it blocks until a client calls; otherwise it serve any waiting call, given it is feasible to do so (e.g., no meaning of getting an element from an empty buffer).

\textbf{The “select” clause:} It facilitates the server to serve more than one client tasks asynchronously and feasibly. It groups a number of accept clauses and places a conditional feasibility check on each, where no clause is to be executed unless the condition is true. Hence, it looks at all of its hosted accept statements and selects those with “true” conditions (“open”) and nonempty mailboxes (“ready”); then arbitrarily (non-deterministically) selects one only to be executed (if more than one are open). If none of the accept clauses are open, or \textbf{open} but with empty mailbox, then the select clause must have either an open delay or terminate clauses to force a wait, at the select entry, for some specified delay or until one of the accept clauses
becomes open and ready, then loops again. The select clause terminates when all depending tasks terminate.