MODULA-2

- The design is centered on the ***modularity*** and ***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, BOLLEAN, 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:

```
VAR A: ARRAY [1..16] of BITSET;
  BEGIN    A[1] := {0-7,9,15} ;
     (* the i^st 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" modules as static scope definers (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.

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 according to the following two scenarios:

1) the name is **exported by a sibling of *M*** (i.e., ***M1***) within its host (i.e., ***N***), and ***M*** **imports** such name.

2) the name is **visible** to the contour (module/procedure) immediately hosting ***M*** (i.e., ***N***) as a locally declared name (to ***N***) or imported name (by ***N***), and ***M*** **imports** such name.

b) An internal module ***M*** can use, in its code, an ***internal*** name only if it is locally declared in ***M*** 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 with many logical types (as long as they are of the same size), thus the compiler can not carry out 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.

- Had the designer of Modula-2 selected the dynamic type checking as a solution, it would have worked, BUT they would have lost in the direction of efficiency (why?).

- 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.

Look the handout example of "generic" ADT stack.

- 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 "logically" compatible with many 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 polymorphic power with the type "any" but we loose security because of the (efficient) static type checking. If the designer of Modula-2 selected the dynamic type checking as a solution, it would work, BUT they would lose in the direction of efficiency (why?).