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 reserved words (BOLD)

Reserved words have special meaning. They cannot be used for any other purpose.

| `abort` | `abs` | `accept` | `access` | `all` | `and` |
| `array` | `at` | `begin` | `body` | `case` | `constant` |
| `declare` | `delay` | `delta` | `digits` | `else` | `elsif` |
| `end` | `entry` | `exception` | `exit` | `for` | `function` |
| `generic` | `goto` | `if` | `in` | `is` | `limited` |
| `loop` | `mod` | `new` | `not` | `null` | `of` |
| `or` | `others` | `out` | `package` | `pragma` | `private` |
| `procedure` | `raise` | `range` | `record` | `rem` | `renames` |
| `return` | `reverse` | `select` | `separate` | `subtype` | `task` |
| `terminate` | `then` | `type` | `use` | `when` | `while` |
| `with` | `xor` |

For their use, see --> http://www.seas.gwu.edu/~adagroup/ada95-syntax/idxkeyword.html

Pre-defined words

Some other words have standard pre-defined meaning. Their meaning can be changed ... with care ... but don't!

| `BOOLEAN` | `CHARACTER` | `CLOSE` | `CREATE` | `DELETE` |
| `FALSE` | `FLOAT` | `GET` | `INTEGER` | `NATURAL` |
| `NEW_LINE` | `OPEN` | `PUT` | `PUT_LINE` | `POSITIVE` |
| `READ` | `RESET` | `SKIP_LINE` | `STRING` | `TEXT_IO` |
| `TRUE` | `WRITE` |
* Ada types as in Pascal & Modula-2, plus:

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

\[
\begin{align*}
A & \quad \text{constant} \quad \text{FLOAT} := 1000.00 ; \\
B & \quad \text{constant} \quad \text{INTEGER} := f(A); \quad \text{-- } f \text{ is a function that returns integer} \\
E_C & \quad \text{constant} \quad := 0.57721 ;
\end{align*}
\]

\[
\text{Light-Year := 5_878_000_000_000 ; -- “named number” for readability}
\]

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 to make programs portable.

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

For **portability** the programmer indicates the required precision:

\[
\begin{align*}
\text{type} \quad \text{DISTANCE} & \quad \text{is} \quad \text{digit} \quad 6 \quad \text{range} \quad 0.0 \ldots 10000.0 \\
\text{-- the programmer specifies his/her a 6 digits of accuracy in the “mantissa” FLOAT type, with range of values 0 to 10000!}
\end{align*}
\]

If the hardware can not provide, the compiler will reject; otherwise it will select one of the above REAL types.

iii) **Subtypes**:

\[
\begin{align*}
\text{type} \quad \text{DAYS} & \quad \text{is} \quad \text{(MON, TUE, WED, THUR, FRI, SAT, SUN)}; \\
\text{subtype} \quad \text{BANKING_DAY} & \quad \text{is} \quad \text{DAYS} \quad \text{range} \quad \text{MON..FRI}; \\
\text{subtype} \quad \text{WEEK_END} & \quad \text{is} \quad \text{DAYS} \quad \text{range} \quad \text{SAT..SUN}; \\
\text{subtype} \quad \text{SHORT_TERM} & \quad \text{is} \quad \text{INTEGER} \quad \text{range} \quad 1..365; \\
\text{subtype} \quad \text{ACCOUNT_BALANCE} & \quad \text{is} \quad \text{DOLLAR} \quad \text{range} \quad 100.00..100_000.00; \\
\text{-- DOLLAR is user defined type} \\
\text{subtype} \quad \text{ACCOUNT_ID} & \quad \text{is} \quad \text{POSITIVE} \quad \text{range} \quad 1..1e9; \\
\text{-- POSITIVE all Integer values starting at 1 through the highest value of Integer} \\
\text{-- NATURAL is a predefined subtype of Integer with a minimum value of 0 and a maximum value of the highest value of Integer .}
\end{align*}
\]

S : SHORT_TERM; J : INTEGER;
J := S; -- OK! (why?)
S := J; -- 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 mainly on the “type” of such entity, leaving values’
constraint 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)

type Daily_Sales is array(DAYS) of FLOAT;

    function Weekly_Total(Shop_Sales : Daily_Sales)
       return Float is
      Sum : Float := 0.0;
    begin
      for index in Shop_Sales’RANGE loop
        Sum := Sum + Shop_Sales(index);
      end loop;
      return Sum;
    end Weekly_Total;
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 “FLOA_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 \( N \) 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, others \( \rightarrow \) 0.0); -- 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,  
-- and their slices must be of equal size.

**Can we do the above array slices' assignment in case with the following**  
**A&B arrays' 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 has 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!

```plaintext
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 implementation of polymorphic procedures and functions.

```plaintext
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”.

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

-- POSITIVE type is defined as all Integer values starting at 1 through
-- the highest value of Integer

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.

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

type PAGE is array (1..500) of STRING (1..80); -- legal
type PAGE is array (1..500) of LINE;
```
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, "TOUGH2DO", 125.00); or
```

*Named association: (for more readability)*

```ada
THIS_TRANSACTION := (ACCOUNT  55_238_1245,
                      PASSWORD  “TOUGH2DO”,
                      BALANCE  125.00);  --
```

**Discriminant Records: (Power of Polymorphism)**

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 “discriminant” mechanism to parameterize the record for any varying field value, while keeping the same record type declaration.

```ada
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:

```ada
A1, A2 : TRANSACTION(3);  -- PSWD of 3 chars
```
B : TRANSACTION (PASSWORD_SIZE \(\rightarrow\) N);
   -- B has N chars password; if N is variable, then its value is not
    known until run time when \textbf{elaborating} the declaration!

\textbf{(dynamic \textit{“password” array})}

\begin{verbatim}
A1 := A2; --ok
A1 := (3, 55_238_1245, “4u2”, 125.00); -- ok
A1 := B -- ok if N is checked to be 3 \textbf{at run time}!
\end{verbatim}

\textbf{Variant Record Facility:}
It is used to declare one record type with variant info for some of its fields,
used to declare records that vary in some of their fields.

\begin{verbatim}
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 \rightarrow
         HOUSEBROKEN : BOOLEAN;
      when CAT \rightarrow
         FUZZY : BOOLEAN;
      when PARROT \rightarrow
         VOCABULARY : NATURAL;
      when others \rightarrow -- it covers for the rest of pets, and must be there!
         null; -- no component exists for such \textit{discriminant value}!
   end case;
end record;
\end{verbatim}

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:

\begin{verbatim}
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!
\end{verbatim}
Remember, the above declarations are all *elaborated* at run-time, thus any misuse (see below) will not be noticed at compile time, instead it will cause “constraint-error” at run time.

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

Question: Are variant records 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: GIANT.KIND:= PARROT;  -- is illegal.
Otherwise, whatever leftover value from the “DOG” GIANT 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 all fields of the record in hand.

**vi) Access Types (Pointers)**

```ada
type TRANSACTION_PTR is access TRANSACTION;

FIRST.Transaction, SECOND.Transaction:
   TRANSACTION_PTR;  -- declare 2 ptrs to trans.
LAST.ZERO := TRANSACTION;  -- record declaration

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, which by the way overloaded to serve also the record field projection. Such overloading will be resolved by the compiler from the context of use, i.e., the type of the name before the “.”.
Assign the fields’ values of an already existing record (LAST_TRANSACTION) to a dynamically allocated record pointed to by SECOND_TRANSACTION:

SECOND_TRANSACTION.all := LAST_TRANSACTION; -- No aliasing
(ptr to TRANSACTION) (TRANSACTION itself)

Assignment of two dynamically allocated records (via new), T1 and T2:
T1.all := T2.all ; -- No aliasing

**Aliasing two pointers to the same record:**
T1 := T2; -- pointer assignment only, T1 and T2 will alias on the same
-- record, originally pointed to by T2 .

- A default value of a pointer is “null”. 