G22.2210

Programming Languages: PL

B. Mishra
New York University.

Lecture # 5

—Slide 1—

The ADA Programming Language
Language Survey 2

- *Countess Ada Augusta Lovelace*
  First programmer (hacker) on Babbage's *Analytical Engine*.
- Relatively new programming Language (1980) developed by DoD. Currently, the only language approved for DoD software project
- Developed to reduce high cost of *designing*, *writing* and *maintaining* DoD software.
- Major items of software are for "**embedded systems**"
  - Primary purpose is control.
  - Incorporated in larger systems.
  - Large software: 50,000–100,000 lines of code
  - Long-lived systems (10–15 years)
  - Continuously evolving.

—Slide 2—

The Design History

- 1975: HOLWG (High Order Language Working Group) was established to investigate problems of developing common language for DoD.
- 1975–1978: List of requirements: *STRAWMAN*, *WOODENMAN*, *TINMAN*, *IRONMAN*, *Revised IRONMAN*, and finally *STEELMAN*
- Design Competition:
  Winner: Jean Ichbiah, Cii Honeywell Bull
- 1987: Ada ISO Standard 8652
- 1983–1987: Analysis of Technical Queries by ARG (Ada Rapporteur Group). Resulted in AIs (Ada Issues).
- 1988: DOD (and ANSI) established Ada 9X project.
- 15 Feb 1995: Ada 95. Core Language + Specialized Annexes

—Slide 3—

Quick Overview (Ada 83)

- Strong Typing
  Scalar, Composite, Access, Private & Derived
- Representation Specification
- Standard Control Constructs:
  Structured Language–Pascalish
- Subprograms: Procedures & Functions
- Program Structuring Facilities: Packages
- Generics: Polymorphisms Extends the concept of type and functional abstractions
- Exceptions
- Separate Compilation
- Tasking (Rendezvous)

—Slide 4—

Quick Overview (Ada 95)

- Object Oriented Programming
- Program Libraries
  (Replaces Ada 83's flat structure by a hierarchy—Visibility Control and Program Extension without recompilation)
- Interfacing
  Graphical User Interface (GUI)—Program Call-back.
- Tasking (Static Monitor)

—Slide 5—

Ada Type System

- Programmer Defined Types
  Set of values, Applicable set of operations and set of properties, accessible via `attributes`
- **Type Definition**

```
type <identifier> is <type-definition>
```

- **Type Binding**

```
<variable> : <type>
```

- **Attributes**

```
<name>´<attribute-identifier>
```


—Slide 6—

Examples

```
type DAY is (MON,TUE,WED,THU,FRI,SAT,SUN);
   --Enumeration Type
TODAY: DAY; TOMORROW: DAY; CURRENT_DAY: DAY;

HOURS_WORKED: array(DAY´FIRST..DAY´LAST)
                   of INTEGER;

if TODAY <= FRI then
   TOMORROW := DAY´SUCC(TODAY);

for CURRENT_DAY in DAY´FIRST..DAY´LAST loop
   ...
end loop;
```


—Slide 7—

Primitive Scalar Types

- Discrete Types
  i) Enumeration Types, ii) Character type, iii) Boolean Types, iv) Integer Types
- Real Types
  i) Fixed-Point Types, ii) Floating-Point Types
- **Note:** Integer and Real types together form the *numeric types.*
- `INTEGER` = Predefined integer types
  `SYSTEM.MIN_INT..SYSTEM.MAX_INT`
- Operation
  `+`, `-`, `*`, `/` (Arithmetic operations)
  `<`, `<=`, `=`, `/=`, `>=`, `>` (Logical operations)

—Slide 8—

Discrete Types

- **Note:** Characters & Boolean are predefined *enumeration types*

  Characters = 128 ASCII characters: 'A', 'B', 'C', ...

  Booleans = `FALSE` & `TRUE`
- All discrete types are **ordered**: `FALSE` < `TRUE`.
- Discrete type values can be used for **indexing** & **iteration over loops**.
- Attributes of Discrete Types (e.g., `T`)

  `T´POS`: Position Number

  ```
  SUIT´POS(HEARTS) = 0
  ```

  `T´VAL`: Inverse of `POS`

  ```
  SUIT´VAL(0) = HEARTS
  ```

  `T´SUCC`, `T´PRED`

  `T´FIRST`, `T´LAST`

—Slide 9—

Real Type

- **Floating Point:**
  Accuracy is fixed by a *relative* error bound

  ```
  type WEIGHT is digits 10;
  ```

  Values have an accuracy of 10 digits.
- **Fixed Point:**
  Accuracy is fixed by a *relative* error bound

  ```
  type VOLTAGE is delta 0.1 range 0.0..1.0;
  ```

  Values have an accuracy at least as fine as 0.1.
- Some Attributes
  `FX` = Fixed Point: $(-2^N+1)\delta..(2^N+1)\delta$

  ```
  FX´DELTA, FX´LARGE
  ```

  `FL` = Floating Point: $\mathrm{sgn} * \mathrm{mantissa} * 2^E$; mantissa has B digits.

  ```
  FL´DIGITS, FL´MANTISSA, FL´EMAX,
  FL´SMALL, FL´LARGE, FL´EPSILON
  ```


—Slide 10—

Derived Type

- Specific Type + *Constraints*
  E.g., Range Constraints for scalar
- Examples

  ```
  subtype BYTE_SIZE is INTEGER range -2**7..2**7;
  subtype CAPS is CHARACTER range ´A´..´Z´;
  subtype <identifier> is <parent-type> range <constraint>;
  ```
- **Object Declaration**

  ```
  X, Y: constant INTEGER range 0..128 = abs(N);
  <identifier>: <mutability> <type> <constraint>
          = <init-value>;
  ```
- Range constraint can be tested at runtime

  ```
  if CURRENT_INPUT not in CAPS then ...
  ```


—Slide 11—

Assignment Statement

```
X := Y;
```

- **Types must match**
  - Same type name–Not structure. Ada uses *name equivalence.*
  - Type checking is at compile time.
  - Type mismatch ⇒
    Program considered *illegal.*
- **Subtype constraints**
  - Compatibility condition: type(X) $\geq$ type(Y)
  - Constraint checking is at run time.
  - Violation raises `constraint` *exception.*
    (Program is **not** illegal.)

—Slide 12—

Examples

```
subtype NATURAL is INTEGER range 1..INTEGER´LAST;
A: INTEGER;
B: FLOAT;
C: NATURAL;
D: INTEGER range 0..INTEGER´LAST;

A := B;              --illegal
A := INTEGER(B);     --type conversion, legal
A := C;              --legal
A := D - 3;          --legal
A := C + INTEGER(B); --legal
C := D;              --constraint exception
C := A;              --constraint exception
```


—Slide 13—

Arrays

- **Arrays in Ada:**
  - Fixed Size—Type may be *unconstrained* at definition But bounds must be given at object declaration.
  - Elements are all of same *subtype*
  - Permits: Assignment, Equality Testing, Slicing, ...

```
subtype NATURAL is INTEGER range 1..INTEGER´LAST;
type SHORT_STRING is array(1..10) of CHARACTER;

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

NAME: STRING(1..10);
```


—Slide 14—

Array Assignment

- Assigning array `B` to `A`: `A := B`
  1. Legal, if type of `A` is same as type of `B`.
  2. If `A` has same number of elements as `B`, then `B` is copied into `A` *positionally*—Otherwise, `constraint-error exception` is raised.

```
declare
  A: STRING(1..10); B: STRING(11..20);
begin
  A := B;
end;
```

- **Array Attributes**

`A´LENGTH`	Number of elements
`A´FIRST`	Lower Bound
`A´LAST`	Upper Bound
`A´RANGE`	subtype `A´FIRST..A´LAST`

—Slide 15—

Array Indexing & Slicing

- Array Indexing:

```
S: STRING(1..10);
S(3) := S(2);
```

- Array Slicing (1D arrays only)

```
S(3..7)              --an array object
S(3..7) := S(4..8);
S := S(2..10) & S(1) -- & == concatenation opn
```

- Array Aggregates:

```
type SYM_TAB is array (CHARACTER range <>) of INTEGER;
TABLE: SYM_TAB(´a´..´c´) := (0, 1, 2);
TABLE := (´c´ => 2, ´b´ => 1, ´a´ => 0);
TABLE := (´c´ | ´b´ => 1, others => 0);
```

—Slide 16—

Records

- **Records in Ada:**
  - *Heterogeneous*: Components need not be of same type.
  - Fields are accessed by component names: E.g., `MY_CAR.CAR_MAKE`
  - *Variant Records* Tag (discriminant) fields cannot be changed at run-time.
  - Permits: Assignment and Equality Testing.

```
type CAR_MAKE is (FORD, GM, HONDA);
subtype CAR_YEAR is INTEGER range 1900..1999;
type CAR is
  MAKE: CAR_MAKE;
  YEAR: CAR_YEAR;
end record;

MY_CAR: CAR;
```


—Slide 17—

Records (Contd)

- Records may be *nested*...*initialized* at declaration.
- A record `B` may be assigned to record `A`, provided they have same type.

```
A, B: CAR;
A := B;
```

- Record Aggregates:

```
YOUR_CAR: CAR :=            YOUR_CAR: CAR :=
   (GM, 1981);                 (MAKE => GM,
                                YEAR => 1981);
```


—Slide 18—

Variant Records

- Similar to PASCAL variant records:
- Except—Type declaration only defines a template; When object is declared, *discriminant value must be supplied*.

```
type VEHICLE_TAG is (CAR, TRUCK);
type VEHICLE(TAG: VEHICLE_TAG) is record
  YEAR: MODEL_YEAR := 93;
  case TAG is
    when CAR => COLORS: COLOR_SCHEME;
    when TRUCK => AXLES: NATURAL;
  end case;
end record;

YOUR_TRUCK: VEHICLE(TRUCK);
REFRIGERATOR: VEHICLE;      --Illegal
```

- There may be more than one discriminant...But they must all be of discrete types....Discriminant can be used as an uninitialized constraint.

```
type BUFFER(LENGTH: NATURAL) is record
  POOL: STRING(1..LENGTH);
end record;
```


—Slide 19—

Access Types

- Allow manipulation of pointers.
- Allow control of object creation.

```
type STRING_PTR is access STRING;
type STRING_10_PTR is access STRING(1..10);
P, Q: STRING_PTR; P10: STRING_10_PTR;

P10 := new STRING(1..10);
P10 := new STRING(2..11);    --Constraint Error
P10 := new STRING;           --Illegal

P := new STRING(1..3);       --OK
P.all := "BUD";
Q := new STRING("MUD");
P := Q;
P.all := Q.all
```

- `new` creates a new object that can be designated by the access type.

—Slide 20—

Recursive Types

```
type NODE;  --Incomplete Declaration;
type NODE_PTR is access NODE;

type NODE is
  record
    DATUM: CHARACTER;
    NEXT: NODE_PTR;
  end record;
```


—Slide 21—

Generalized Access Types

- Inherent Access to declared objects
  (Not just objects created by allocators)

```
type INT_PTR is access all INTEGER;

IP: INT_PTR;
I: aliased INTEGER;
IP := I'Access
```

- **Note:** Designated variable must be marked `aliased`.
- `Access` attribute is only applicable to an object whose lifetime is at least that of the access type.
- Avoids "dangling reference" problem.

```
type CONST_INT_PTR is access constant INTEGER;

CIP: CONST_INT_PTR;
C: aliased constant INTEGER := 1815;
CIP := C'Access
```

- Access is restricted to read-only

—Slide 22—

Control Structures

- Assignment Statements

```
DISCRIM := (B**2 - 4.0*A*C);
TABLE(J) := TABLE(J) + 1;
VECTOR := (1..10 => 0);
```

- Conditional Statements

```
if (A=1) then
  ...
end if;

if (A=1) then
  --...
elsif (A=2) then
  --...
else
  --...
end if;
```

```
case A is
  when 1 => --...;
  when 2 => --...;
  when others => null;
end case;
```


—Slide 23—

Control Structures: Iteration Clause

- Iteration Statements—Basic Loop

```
loop
  -- Statements to be repeated
end loop;
```

- *Iteration Clause*
  Execution of a basic `loop` terminates when
  1. *The iteration is completed* or
  2. *A* `loop exit` *statement is executed*

```
SUM := 0;
for I in 1..10 loop
 SUM := SUM + A(I);
end loop;
```

```
SUM := 0;
for I in reverse 1..10 loop
  SUM := SUM + A(I);
end loop;
```

```
SUM := 0; I := 1;
while I <= 10 loop
  SUM := SUM + A(I);
  I := I + 1;
end loop;
```

```
SUM := 0; I := 1;
loop
  exit when I > 10;
  SUM := SUM + A(I);
  I := I + 1;
end loop;
```

—Last Slide—

A Complete Ada Program

```
with I_O_PACKAGE;
procedure TEMPERATURE_CONVERSION is
  use I_O_PACKAGE;
  -- Convert temp in Fahrenheit to Celsius

  FAHRENHEIT_TEMP; CELSIUS_TEMP: FLOAT;
begin
  GET(FAHRENHEIT_TEMP);
  CELSIUS_TEMP := (FAHRENHEIT_TEMP - 32.0)*5.0/9.0;
  PUT(CELSIUS_TEMP);
end TEMPERATURE_CONVERSION;
```

[End of Lecture #5]