Types

all types
  elementary
    scalar
      discrete
        enumeration
          character
          boolean
          other enumeration
        integer
          signed integer
          modular integer
    ...

History

"Modula-2 — 90% of the power of Ada at 10% of the cost."

- DoD issued a competition to design a language that would take over from the hundred of languages that the military was programming in at the time.
- It is very similar in flavor to Modula-2 but infinitely more complex.
- Ada is a Kitchen-Sink Language.
- It took many years to develop working compilers. In the mean time C++ entered the stage.
Constant Declarations

Limit : constant Integer := 10_000;
Low_Limit : constant Integer := Limit/10;
Tolerance : constant Real := Dispersion(1.15);

Derived Type Declarations

-- two different types:
type Local_Coordinate is new Coordinate;
type Midweek is new Day range Tue .. Thu;
-- same range as Positive:
type Counter is new Positive;

Subtype Declarations

type Color is (White,Red,Yellow,Green,Blue,Brown,Black);
type Column is range 1 .. 72;
type Table is array(1 .. 10) of Integer;
--Examples of subtype declarations:
subtype Rainbow is Color range Red .. Blue;
subtype Red_Blue is Rainbow;
subtype Int is Integer;
subtype Small_Int is Integer range -10 .. 10;
subtype Up_To_K is Column range 1 .. K;
subtype Square is Matrix(1 .. 10, 1 .. 10);
subtype Male is Person(Sex => M);

class all types
elementary
....
composite
array
  string
other array
untagged record
tagged
task
protected
### Fixed Point Type Declarations

```
type Volt is delta 0.125 range 0.0 .. 255.0;
-- A pure fraction which requires all the available
-- space in a word can be declared as the type Fraction:
type Fraction is delta System.Fine_Delta range -1.0 .. 1.0;
-- Fraction’Last = 1.0 - System.Fine_Delta

-- decimal fixed point:
type Money is delta 0.01 digits 15;
subtype Salary is Money digits 10;
```

### Enumeration Type Declarations

```
type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
type Suit is (Clubs, Diamonds, Hearts, Spades);
type Color is (White, Red, Yellow, Green, Blue, Brown, Black);
-- Red and Green are overloaded:
type Light is (Red, Amber, Green);
type Hexa is ('A', 'B', 'C', 'D', 'E', 'F');
type Mixed is ('A', 'B', '*', B, None, '?', '%');

subtype Weekday is Day range Mon .. Fri;
subtype Major is Suit range Hearts .. Spades;
-- the Color Red, not the Light:
subtype Rainbow is Color range Red .. Blue;
```

### Array Type Declarations

```
-- Unconstrained array definitions:
type Vector is array(Integer range <>) of Real;
type Matrix is array(Integer range <>, Integer range <>) of Real;
type Bit_Vector is array(Integer range <>) of Boolean;
type Roman is array(Positive range <>) of Roman_Digit;

-- Constrained array definitions:
type Table is array(1 .. 10) of Integer;
type Schedule is array(Day) of Boolean;
type Line is array(1 .. Max_Line_Size) of Character;
```

### Floating Point Type Declarations

```
type Coefficient is digits 10 range -1.0 .. 1.0;
type Real is digits 8;
type Mass is digits 7 range 0.0 .. 1.0E35;
-- a subtype with a smaller range:
subtype Probability is Real range 0.0 .. 1.0;
```
Discriminated Type Declarations

type Buffer(Size : Buffer_Size := 100) is
  record
    Pos : Buffer_Size := 0;
    Value : String(1 .. Size);
  end record;

type Matrix_Rec(Rows, Columns : Integer) is
  record
    Mat : Matrix(1 .. Rows, 1 .. Columns);
  end record;

type Square(Side : Integer) is new Matrix_Rec(Rows => Side, Columns => Side);

type Double_Square(Number : Integer) is
  record
    Left : Square(Number);
    Right : Square(Number);
  end record;

type Item(Number : Positive) is
  record
    Content : Integer;
    -- no component depends on the discriminant
  end record;

Grid : array(1 .. 80, 1 .. 100) of Boolean;
Mix : array(Color range Red..Green) of Boolean;
-- an array of arrays:
Page : array(Positive range <> ) of Line :=
  (1 | 50 =>
    Line'(1 | Line'Last => '+', others => '-'),
  2 .. 49 =>
    Line'(1 | Line'Last => '|', others => ' '));

-- Index constraint:
Board : Matrix(1 .. 8, 1 .. 8);
Rectangle : Matrix(1 .. 20, 1 .. 30);

-- N need not be static:
Inverse : Matrix(1 .. N, 1 .. N);

Record Type Declarations

  type Date is
    record
      Day : Integer range 1 .. 31;
      Month : Month_Name;
      Year : Integer range 0 .. 4000;
    end record;

  type Complex is
    record
      Re : Real := 0.0;
      Im : Real := 0.0;
    end record;
**Variant Record Type Declarations**

type Device is (Printer, Disk, Drum);
type State is (Open, Closed);

type Peripheral(Unit : Device := Disk) is record
  Status : State;
  case Unit is
    when Printer =>
      Line_Count : Integer range 1 .. Page_Size;
    when others =>
      Cylinder : Cylinder_Index;
      Track : Track_Number;
  end case;
end record;

subtype Drum_Unit is Peripheral(Drum);
subtype Disk_Unit is Peripheral(Disk);

Writer : Peripheral(Unit => Printer);
Archive : Disk_Unit;

**Pointer Type Declarations**

type Cell; -- incomplete type declaration
type Link is access Cell;

type Cell is
  record
    Value : Integer;
    Succ : Link;
    Pred : Link;
  end record;

Head : Link := new Cell'(0, null, null);
Next : Link := Head.Succ;

--Examples of mutually dependent access types:
type Person(<>); -- incomplete type declaration
type Car; -- incomplete type declaration

type Person_Name is access Person;
type Car_Name is access all Car;

type Car is
  record
    Number : Integer;
    Owner : Person_Name;
  end record;
type Person(Sex : Gender) is
record
Name : String(1 .. 20);
Birth : Date;
Age : Integer range 0 .. 130;
Vehicle : Car_Name;
case Sex is
when M => Wife : Person_Name(Sex => F);
when F => Husband : Person_Name(Sex => M);
end case;
end record;

My_Car, Your_Car, Next_Car : Car_Name := new Car;
George : Person_Name := new Person(M);
George.Vehicle := Your_Car;

-- Record Aggregates

-- Record aggregate with positional associations:
(4, July, 1776)

-- Record aggregates with named associations:
(Day => 4, Month => July, Year => 1776)
(Month => July, Day => 4, Year => 1776)
(Disk, Closed, Track => 5, Cylinder => 12)
(Unit => Disk, Status => Closed, Cylinder=>9, Track=>1)

-- Component association with several choices:
(Value => 0, Succ|Pred => new Cell'(0, null, null))

-- Array Aggregates

-- two-dimensional:
(1 .. 5 => (1 .. 8 => 0.0))

-- N new cells, in particular for N = 0:
(1 .. N => new Cell)

Table'(2 | 4 | 10 => 1, others => 0)
Schedule'(Mon .. Fri => True, others => False)
Schedule'(Wed | Sun => False, others => True)

-- single-component vector:
Vector'(1 => 2.5)

-- Array Slices

Stars(1 .. 15) -- a slice of 15 characters
Page(10 .. 10 + Size) -- a slice of 1 + Size components
Page(L)(A .. B) -- a slice of the array Page(L)
Stars(1 .. 0) -- a null slice
My_Schedule(Weekday) -- bounds given by subtype
Stars(5 .. 15)(K) -- same as Stars(K)
-- provided that K is in 5 .. 15
### Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
<td>real literal</td>
</tr>
<tr>
<td>Pi</td>
<td>named number</td>
</tr>
<tr>
<td>(1 .. 10 =&gt; 0)</td>
<td>array aggregate</td>
</tr>
<tr>
<td>Sum</td>
<td>variable</td>
</tr>
<tr>
<td>Integer'Last</td>
<td>attribute</td>
</tr>
<tr>
<td>Sine(X)</td>
<td>function call</td>
</tr>
<tr>
<td>Color'(Blue)</td>
<td>qualified expression</td>
</tr>
<tr>
<td>Real(M*N)</td>
<td>conversion</td>
</tr>
<tr>
<td>(Line_Count + 10)</td>
<td>parenthesized expression</td>
</tr>
</tbody>
</table>

### Labeled Statements, Goto Statement

```
<<Sort>>
for I in 1 .. N-1 loop
  if A(I) > A(I+1) then
    Exchange(A(I), A(I+1));
    goto Sort;
  end if;
end loop;
```
**If Statement**

if Month = December and Day = 31 then
    Month := January;
    Day := 1;
    Year := Year + 1;
end if;

if Line_Too_Short then
    raise Layout_Error;
elsif Line_Full then
    New_Line; Put(Item);
else
    Put(Item);
end if;

**Assignment Statement**

Value := Max_Value - 1;
Shade := Blue;

Next_Frame(F)(M, N) := 2.5;
U := Dot_Product(V, W);

-- Scalar subtype conversions:
I, J : Integer range 1 .. 10 := 5;
K : Integer range 1 .. 20 := 15;
I := J; -- identical ranges
K := J; -- compatible ranges
J := K; -- will raise Constraint_Error if K > 10

**Case Statement**

case Sensor is
    when Elevation => Record_Elevation(Sensor_Value);
    when Azimuth  => Record_Azimuth (Sensor_Value);
    when Distance => Record_Distance (Sensor_Value);
    when others   => null;
end case;

case Today is
    when Mon    => Compute_Initial_Balance;
    when Fri    => Compute_Closing_Balance;
    when Tue .. Thu => Generate_Report(Today);
    when Sat .. Sun => null;
end case;

**Assignment Statement**

Writer := (Status=>Open, Unit=>Printer, Line_Count=>60);
Next_Car.all := (72074, null);

-- Array subtype conversions:
A : String(1 .. 31);
B : String(3 .. 33);
A := B; -- same number of components
A(1 .. 9) := "tar sauce";
A(4 .. 12) := A(1 .. 9); -- A(1 .. 12) = "tartar sauce"
case Bin_Number(Count) is
    when 1 => Update_Bin(1);
    when 2 => Update_Bin(2);
    when 3 | 4 =>
        Empty_Bin(1);
        Empty_Bin(2);
    when others => raise Error;
end case;

-- Loop statement with a for iteration scheme:
for J in Buffer'Range loop   -- works with a null range
    if Buffer(J) /= Space then
        Put(Buffer(J));
    end if;
end loop;

-- Loop statement with a name:
Summation:
    while Next /= Head loop
        Sum := Sum + Next.Value;
        Next := Next.Succ;
    end loop Summation;

loop
    Get(Current_Character);
    exit when Current_Character = '*';
end loop;

-- Loop statement with a while iteration scheme:
while Bid(N).Price < Cut_Off.Price loop
    Record_Bid(Bid(N).Price);
    N := N + 1;
end loop;

Block Statement

Swap:
    declare
        Temp : Integer;
    begin
        Temp := V; V := U; U := Temp;
    end Swap;

Loop Statement
Exit Statement

for N in 1 .. Max_Num_Items loop
    Get_New_Item(New_Item);
    Merge_Item(New_Item, Storage_File);
    exit when New_Item = Terminal_Item;
end loop;

Main_Cycle:
    loop
        -- initial statements
        exit Main_Cycle when Found;
        -- final statements
    end loop Main_Cycle;

Subprogram Declarations

procedure Traverse_Tree;
procedure Increment(X : in out Integer);
procedure Right_Indent(Margin : out Line_Size);
procedure Switch(From, To : in out Link);
function Random return Probability;
function Min_Cell(X : Link) return Cell;
function Next_Frame(K : Positive) return Frame;
function Dot_Product(Left, Right : Vector) return Real;
function "*"(Left, Right : Matrix) return Matrix;

Examples of in parameters with default expressions:

procedure Print_Header(Pages : in Natural;
    Header : in Line := (1 .. Line'Last => ' ');
    Center : in Boolean := True);

Subprogram Bodies

-- Example of procedure body:

procedure Push(E : in Element_Type; S : in out Stack) is
    begin
        if S.Index = S.Size then
            raise Stack_Overflow;
        else
            S.Index := S.Index + 1;
            S.Space(S.Index) := E;
        end if;
    end Push;
-- Example of a function body:

```haskell```
function Dot_Product(Left, Right : Vector) return Real is
  Sum : Real := 0.0;
begin
  Check(Left'First = Right'First and
         Left'Last = Right'Last);
  for J in Left'Range loop
    Sum := Sum + Left(J)*Right(J);
  end loop;
  return Sum;
end Dot_Product;
```

Procedure Call

```haskell```
Traverse_Tree;
Print_Header(128, Title, True);
Switch(From => X, To => Next);
Print_Header(128, Header => Title, Center => True);
Print_Header(Header=>Title, Center=>True, Pages=>128);

--Examples of function calls:
Dot_Product(U, V)
Clock

Overloaded Calls

```haskell```
procedure Put(X : in Integer);
procedure Put(X : in String);

procedure Set(Tint : in Color);
procedure Set(Signal : in Light);

--Examples of their calls:
Put(28);
Put("no possible ambiguity here");
Set(Tint => Red); -- Set(Red) would be ambiguous.
Set(Signal => Red); -- Red can denote either a Color
Set(Color'(Red)); -- or a Light
```

-- Procedures with default expressions:
```
procedure Activate(Process : in Process_Name;
                   After : in Process_Name:=No_Process;
                   Wait : in Duration := 0.0;
                   Prior : in Boolean := False);

procedure Pair(Left, Right : in Person_Name:=new Person);
```

-- Examples of their calls:
```
Activate(X);
Activate(X, After => Y);
Activate(X, Wait => 60.0, Prior => True);
Activate(X, Y, 10.0, False);
Pair;
Pair(Left => new Person, Right => new Person);
```
Userdefined Operators

function + (Left, Right : Matrix) return Matrix;
function + (Left, Right : Vector) return Vector;

-- assuming that A, B, and C are of the type Vector
-- the following two statements are equivalent:
A := B + C;
A := "+"(B, C);

Package Declaration

package Rational_Numbers is
    type Rational is
        record
            Numerator : Integer;
            Denominator : Positive;
        end record;
    function = (X,Y : Rational) return Boolean;
end Rational_Numbers;

Package Body Declaration

package body Rational_Numbers is
    procedure Same_Denominator (X,Y : in out Rational) is
        begin
        end Same_Denominator;
    function = (X,Y : Rational) return Boolean is
        U : Rational := X;
        V : Rational := Y;
        begin
            Same_Denominator (U,V);
            return U.Numerator = V.Numerator;
        end =;
end Rational_Numbers;

-- to construct a rational number:
function / (X,Y : Integer) return Rational;
function + (X,Y : Rational) return Rational;
function - (X,Y : Rational) return Rational;
function * (X,Y : Rational) return Rational;
function / (X,Y : Rational) return Rational;
end Rational_Numbers;
package body Key_Manager is
  Last_Key : Key := Null_Key;
procedure Get_Key(K : out Key) is
begin
  Last_Key := Last_Key + 1;
  K := Last_Key;
end Get_Key;

function "<" (X, Y : Key) return Boolean is
begin
  return Natural(X) < Natural(Y);
end "<";
end Key_Manager;

renaming

procedure My_Write(C : in Character)
  renames Pool(K).Write;

function Real_Plus(Left, Right : Real ) return Real
  renames "+";
function Int_Plus (Left, Right : Integer) return Integer
  renames "+";

function Rouge return Color renames Red;
function Rot return Color renames Red;
function Rosso return Color renames Rouge;

package Key_Manager is
  type Key is private;
begin
  -- a deferred constant declaration:
  Null_Key : constant Key := Natural'Last;
procedure Get_Key(K : out Key);
function "<" (X, Y : Key) return Boolean;

end Key_Manager;
Exceptions

Singular : exception;
Error : exception;
Overflow, Underflow : exception;

begin
  Open(File, In_File, "input.txt");
exception
  when E : Name_Error =>
    Put("Cannot open input file : ");
    Put_Line(Exception_Message(E));
    raise; -- re-raise the current exception
end;

Generic Declarations

generic
  type Elem is private;
  procedure Exchange(U, V : in out Elem);

generic
  type Item is private;
  with function "*"(U, V : Item) return Item is <>;
  function Squaring(X : Item) return Item;

Separate Compilation

package body Parent is
    Variable : String := "Hello, there.";
    procedure Inner is separate;
end Parent;

with Ada.Text_IO;
separate(Parent)
procedure Inner is
begin
    Ada.Text_IO.Put_Line(Variable);
end Inner;
package body On_Vectors is

  function Sum(A, B: Vector) return Vector is
    Result : Vector(A'Range); -- the formal type Vector
    Bias   : constant Integer := B'First - A'First;
    begin
      if A'Length /= B'Length then
        raise Length_Error;
      end if;

      for N in A'Range loop
        -- the formal function Sum:
        Result(N) := Sum(A(N), B(N + Bias));
      end loop;
      return Result;
    end Sum;

  function Sigma(A : Vector) return Item is
    Total : Item := A(A'First); -- the formal type Item
    begin
      for N in A'First + 1 .. A'Last loop
        Total := Sum(Total, A(N));
      end loop;
      return Total;
    end Sigma;
  end On_Vectors;

procedure Exchange(U, V : in out Elem) is
  T : Elem; -- the generic formal type
  begin
    T := U;
    U := V;
    V := T;
  end Exchange;

function Squaring(X : Item) return Item is
  begin
    return X*X; -- the formal operator "*"
  end Squaring;

-- Generic declaration declaring a generic package:
generic
  type Item is private;
  type Vector is array (Positive range <>) of Item;
  with function Sum(X, Y : Item) return Item;
package On_Vectors is
  function Sum (A, B: Vector) return Vector;
  function Sigma(A : Vector) return Item;
  Length_Error : exception;
end On_Vectors;

Generic Bodies
A Generic Package

- An Ada module specification has two parts, a public part and a private part.
- The private part contains the definitions of all those items that we don’t want a user to know about. In this example, the private part reveals that the stack is implemented as an array.

generic
  type ITEM is private;
package GENERIC_STACK is
  type STACK (SIZE : POSITIVE) is limited private;
  procedure PUSH (S : in out STACK; E : in ITEM);
  procedure POP (S : in out STACK; E : out ITEM);
  pragma INLINE (PUSH, POP);
private
  type VECTOR is array (POSITIVE range < >) of ITEM;
  type STACK (SIZE : POSITIVE) is record
    SPACE : VECTOR (1 .. SIZE);
    INDEX : NATURAL := 0;
  end record;
end GENERIC_STACK;

Generic Instantiations

procedure Swap is new Exchange(Elem => Integer);
procedure Swap is new Exchange(Character);
-- Swap is overloaded
function Square is new Squaring(Integer);
-- "*" of Integer used by default
function Square is new
  Squaring(Item => Matrix, "*" => Matrix_Product);
function Square is new
  Squaring(Matrix, Matrix_Product); -- same as previous
package Int_Vectors is new On_Vectors(Integer, Table,"+");

--Examples of uses of instantiated units:
Swap(A, B);
A := Square(A);
T : Table(1 .. 5) := (10, 20, 30, 40, 50);
package body GENERIC_STACK is
-- Implementations of ...
-- PUSH and POP ...
end GENERIC_STACK;

with GENERIC_STACK;
procedure MAIN is
package STACK_INT is new GENERIC_STACK (INTEGER);

S : STACK_INT.STACK (100);
begin
STACK_INT.PUSH (S, 314);
end MAIN;

type Mix_Code is (ADD, SUB, MUL, LDA, STA, STZ);
for Mix_Code use
  (ADD => 1, SUB => 2, MUL => 3,
   LDA => 8, STA => 24, STZ =>33);

type State is (A,M,W,P);
type Mode is (Fix, Dec, Exp, Signif);

type Byte_Mask is array (0..7) of Boolean;
type State_Mask is array (State) of Boolean;
type Mode_Mask is array (Mode) of Boolean;

Word : constant := 4;  -- storage element is byte, 4 bytes per word

type Medium is range 0 .. 65_000;
for Medium'Size use 2*Byte;
for Medium'Alignment use 2;
Device_Register : Medium;
for Device_Register'Size use Medium'Size;
for Device_Register'Address use
  System.Storage_Elements.To_Address(16#FFFF_0020#);

type Short is delta 0.01 range -100.0 .. 100.0;
for Short'Size use 15;

Byte : constant := 8;
Page : constant := 2**12;

for Medium use
  (ADD => 1, SUB => 2, MUL => 3,
   LDA => 8, STA => 24, STZ =>33);
**Interface to Other Languages**

package Fortran_Library is
function Sqrt (X : Float) return Float;
function Exp (X : Float) return Float;
private
pragma Import(Fortran, Sqrt);
pragma Import(Fortran, Exp);
end Fortran_Library;

**Readings and References**

- http://www.informatik.uni-stuttgart.de/ifi/bs/lehre/ei1/adanotes
- http://www.cs.fit.edu/~ryan/ada
- An Ada95 compiler that generates Java bytecode: http://www1.acm.org/sigs/sigada/education/pages/jgnat.html
- The Ada Reference Manual (ARM): http://www.ada-auth.org/~acats/arm.html. The examples in this lecture are taken from the ARM.

**type Program_Status_Word is**

record
System_Mask : Byte_Mask;
Protection_Key : Integer range 0 .. 3;
Machine_State : State_Mask;
Interrupt_Cause : Interruption_Code;
Ilc : Integer range 0 .. 3;
Cc : Integer range 0 .. 3;
Program_Mask : Mode_Mask;
Inst_Address : Address;
end record;

**for Program_Status_Word use**

record
System_Mask at 0*Word range 0 .. 7;
Protection_Key at 0*Word range 10 .. 11;
-- bits 8,9 unused
Machine_State at 0*Word range 12 .. 15;
Interrupt_Cause at 0*Word range 16 .. 31;
-- second word:
Ilc at 1*Word range 0 .. 1;
Cc at 1*Word range 2 .. 3;
Program_Mask at 1*Word range 4 .. 7;
Inst_Address at 1*Word range 8 .. 31;
end record;
Compiling Ada

• The jgnat compiler has been installed on lectura. Do the following:
  – setenv PATH other
    stuff:/home/cs520/2003/ada/jgnat-1.1p/bin
  – setenv CLASSPATH other
    stuff:/home/cs520/2003/ada/jgnat-1.1p/lib/jgnat.jar

  jgnat compiles Ada to Java classfiles.

• jgnat can be downloaded from

> setenv PATH ...:/home/cs520/2003/ada/jgnat-1.1p/bin
> cat > hello.adb
  with Ada.Text_IO; use Ada.Text_IO;
  procedure Hello is
  begin
    Put_Line ("hello JGNAT");
    end Hello;
> jgnatmake hello
  jgnat -c hello.adb
  jgnatbind -x hello.ali
  jgnatlink hello.ali
> setenv CLASSPATH \ 
  .:/home/cs520/2003/ada/jgnat-1.1p/lib/jgnat.jar
> java hello
  hello JGNAT