



Ada in a Nutshell

- make Ada in 180 minutes -

2002–2004© Departments Programming Languages
& Software Engineering
Institute of Computer Science
University of Stuttgart

Table of Contents

| | |
|---|----|
| Overview | 3 |
| Example: Straight Insertion Sorting | 5 |
| Compilation Unit | 11 |
| Packages | 12 |
| Subprograms | 25 |
| Overloading | 36 |
| Statements | 41 |
| Types | 50 |

Overview

- ❑ background of Ada
- ❑ first steps and the Gnu Ada compiler *gnat*
- ❑ packages
- ❑ subprograms and overloading
- ❑ statements and expressions
- ❑ types

Features of Ada

Ada supports

- ❑ **abstraction** and **information hiding** by virtue of modules (packages)
- ❑ **re-usability** by means of generic units and inheritance
- ❑ **reliability** by means of strict type checking and exception handling
- ❑ **maintenance** by means of locality and uniform concepts
- ❑ **parallelism** and **synchronization** by means of tasks, rendezvous concepts, and protected types
- ❑ **low-level programming** and **interfacing** with other languages by means of representation clauses for data structures

Example: Straight Insertion Sorting

Algorithm

- The i 'th run moves the i 'th element $A(i)$ at its place within the sequence of elements $A(1), A(2), \dots, A(i-1)$ which has already been sorted by previous runs

Example

| | | | | | | |
|---|----|----|----|----|----|----|
| 5 | 33 | 12 | 13 | 8 | 1 | 41 |
| 5 | 33 | 12 | 13 | 8 | 1 | 41 |
| 5 | 12 | 33 | 13 | 8 | 1 | 41 |
| 5 | 12 | 13 | 33 | 8 | 1 | 41 |
| 5 | 8 | 12 | 13 | 33 | 1 | 41 |
| 1 | 5 | 8 | 12 | 13 | 33 | 41 |

Straight Insertion Sorting (2)

Type declarations

```
subtype Index_Type    is NATURAL range 1..100;
```

```
subtype Element_Type  is NATURAL;
```

```
type Field_Type       is array (Index_Type) of Element_Type;
```

Variable declaration

```
My_Field: Field_Type;
```

Straight Insertion Sorting (3)

```
procedure Straight_Insertion (F: in out Field_Type) is
    j    : Index_Type;
    Temp: Element_Type;
begin
    for i in F'First+1..F'Last loop
        Temp := F(i); j := i;
        while j > F'First and then Temp < F(j-1) loop
            F(j) := F(j-1);
            j := j-1;
        end loop;
        F(j) := Temp;
    end loop;
end Straight_Insertion;
```

the main program

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Sorting is
    ... -- all other declarations
    procedure Straight_Insertion(F : in out Field_Type) is
        ...
    procedure Output (F : in Field_Type) is ...

begin
    ...
    Straight_Insertion (My_Field);

    Put_Line ("sorted field:");
    Output (My_Field);
end Sorting;
```

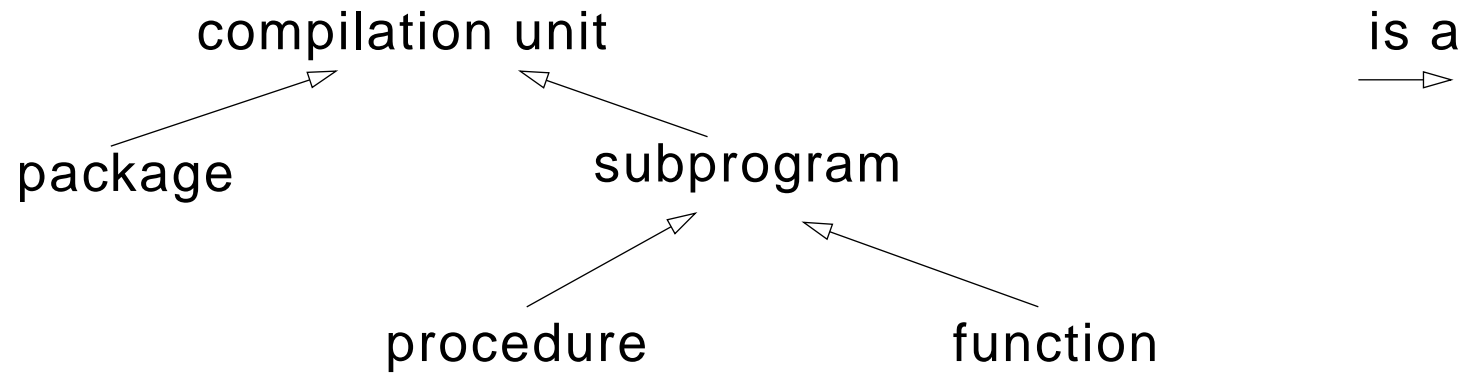
output procedure

```
with Ada.Text_IO;  
with Ada.Integer_Text_IO;  
-- Importing the necessary standard input/output packages  
...  
  
procedure Output (F: in Field_Type) is  
  
begin -- Output  
    for i in F'Range loop  
        Ada.Integer_Text_IO.Put (F(i), width => 4);  
        Ada.Text_IO.New_Line;  
    end loop;  
    Ada.Text_IO.New_Line;  
end Output;
```

compiling the program

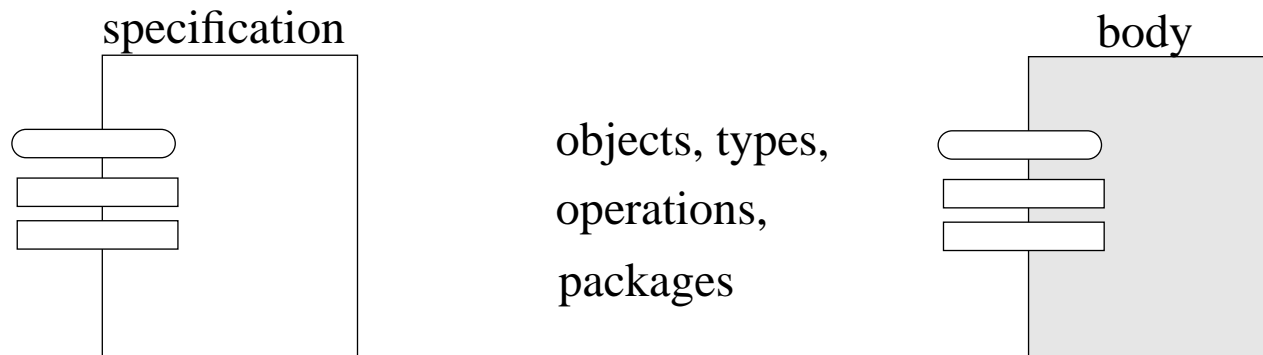
- ❑ the free Ada95 compiler GNAT is part of the Gnu compiler suite *gcc* (yet, is not included in the standard distribution)
- ❑ *gcc* recognizes the suffixes *.ads* and *.adb* and calls the Ada compiler *gnat1*
- ❑ compilation with *gcc -c hello.adb*
 - ⇒ *hello.ali* and *hello.o* will be generated
- ❑ you better use *gnatmake* which will also take care of all necessary compilations of imported packages
 - ⇒ compile: *gnatmake -c hello*
 - ⇒ compile & link: *gnatmake hello*
- ❑ get a free compiler and find other useful links at
 - ⇒ <http://www.iste.uni-stuttgart.de/ps/ada-doc/>

Compilation Unit



Packages

- ❑ unit for related declarations (constants, global variables, types, subprograms, nested packages)
- ❑ packages come in two parts:
 - ⇒ specification (external interface) and
 - ⇒ body (implementation)
- ❑ supports modularity/grouping, abstraction and locality, information hiding



Gnat-Convention

- ❑ files contain either
 - ⇒ specification of exactly one compilation unit (at top-most level)
 - ⇒ or body of exactly one compilation unit (at top-most level)
- ❑ file with specification has suffix *.ads*
- ❑ file with body has suffix *.adb*
- ❑ example:

my_package.ads

```
package My_Package
is ... end
```

my_package.adb

```
package body My_Package
is ... end
```

Packages in Ada 95

package specification

- ❑ package specification has two parts:
 - ⇒ visible part (visible for clients of the package)
 - ⇒ private part (invisible for clients of the package; human readable, however)
- ❑ put declarations you want to export in the visible part
- ❑ put declarations you want to hide from clients, yet may not be put in the body, in the private part (e.g., concrete type representation, see below)
 - ⇒ supports separate compilation
 - ⇒ plays an important role for subsystems (child units; not discussed here)

Packages in Ada 95

example package specification

```
package Complex is
  type Number is private;
  procedure Set (X: out Number;
                Real_Part: in Float;
                Imaginary_Part: in Float);
  function "+" (X, Y: in Number) return Number;
  function "-" (X, Y: in Number) return Number;

private
  type Number is
    record
      Real_Part: Float;
      Imaginary_Part: Float;
    end record;
end Complex;
```

Packages in Ada 95

primitive subprograms

- ❑ A subprogram U is called *primitive* to a type T if
 - ⇒ T and U are declared in the same package specification and
 - ⇒ T is mentioned in the signature of U (parameter or result type)
- ❑ primitive subprograms are relevant for visibility and inheritance

Packages in Ada 95

visibility of packages (1)

```
procedure Main_Program is
  procedure First is ... end First;
  package Complex is ... end Complex;
  package body Complex is ... end Complex;
  procedure Second is ... end Second;

begin
  -- statement sequence
end Main_Program;
```

- ❑ Complex is visible within Main_Program
- ❑ scope starts at first occurrence, i.e., Complex is not visible for First but for Second

Packages in Ada 95

visibility of packages (2)

```
with Complex; use Complex;
procedure Main_Program is
    Number_1, Number_2: [Complex.]Number;
begin
    ... -- statement sequence
end Main_Program;
```

- ❑ use `with` to import package `Complex`; you can refer to declarations in `Complex` (visible part of the specification) using qualified notation, e.g., `Complex.Number`
- ❑ `use Complex` makes all declarations *directly* visible
- ❑ `use type Complex.Complex` make only the primitive operators (+, -, *, etc.) of type (but not the type itself) directly visible

Packages in Ada 95

package bodies

- ❑ name of the body must conform to the name used in the specification
- ❑ declarations in the body cannot be accessed from outside
- ❑ body may contain
 - ⇒ declarative part
 - ⇒ sequence of statements (for initialization purposes, implicitly called during elaboration on system start-up)

Packages in Ada 95

Example of a package body (1)

```
package body Complex is
  procedure Set (X: out Number;
                Real_Part: in Float;
                Imaginary_Part: in Float) is
  begin
    X := (Real_Part, Imaginary_Part);
  end Set;

  function "+" (X, Y: in Number) return Number is
  begin
    return (X.Real_Part + Y.Real_Part,
            X.Imaginary_Part + Y.Imaginary_Part);
  end "+";
  ...
end Complex;
```

Packages in Ada 95

Example of a package body (2)

```
package Random is
    function Number return Float;
end Random;
```

} Specification

```
package body Random is
    Seed: Integer;
    function Number return Float is
        ...
    end Number;
begin
    Seed := 1234567;
end Random;
```

} body
} initialization

Packages in Ada 95

private types in Ada (1)

- ❑ known: package specification consists of visible and private part
- ❑ new: there are two kinds of private types
 - ⇒ simple private types (nonlimited private)
 - ⇒ limited private types
- ❑ operations applicable to nonlimited types:
 - ⇒ explicitly declared subprograms in package specification, assignment, tests “=” and “/=”
- ❑ operations applicable to limited private types:
 - ⇒ as above, but no assignment
 - ⇒ gives programmer more control

Nonlimited Private Type

```
package Blocks is
  type Block is private;
  procedure New_Block return Block;
  procedure Fill (B : Block; D: Data);
  procedure Release (B : in out Block);
private
  type Block is access Data;
end Blocks;

with Blocks; use Blocks;
procedure Main is
  B1, B2 : Block;
begin
  B1 := New_Block;
  B2 := B1;
  Release (B1);
  Fill (B2, D); -- Ouch
end Main;
```



Limited Private Type

```
package Blocks is
  type Block is limited private;
  procedure Copy (From : in Block; To : out Block);
  ...
end Blocks;

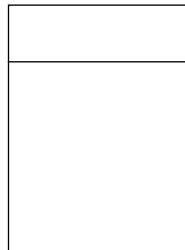
with Blocks; use Blocks;
procedure Main is
  B1, B2 : Block;
begin
  B1 := New_Block;
  B2 := B1; -- not allowed, checked by compiler
  Copy (B1, B2);
  Release (B1);
  Fill (B2, D); -- OK
end Main;
```

Subprograms

Subprograms are

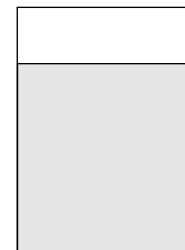
- ❑ procedures
- ❑ or functions
 - ⇒ functions return a single result

specification



(optional)

body



symbols for subprograms

Subprograms

| | | |
|---|-------------------------------------|--|
| <pre>procedure Name (parameters);</pre> | Subprogram Specification | <pre>function Name (parameters) return Type_Name;</pre> |
| <pre>procedure Name (parameters) is -- local declarations begin -- statements end Name;</pre> | Subprogram Body | <pre>function Name (parameters) return Type_Name is -- local declarations begin -- statements -- must include return end Name;</pre> |

- ❑ generally, every subprogram has a specification and a body
- ❑ body contains declarations and statements

Formal Parameter Modes

- ❑ Formal parameters may have one of three modes:
`in, out or in out`
 - ⇒ `in`: the parameter may be read but not be changed (default)
 - ⇒ `out`: the parameter may be changed (may also be read but is initially undefined); the value of the formal out-parameter is assigned to the actual parameter upon procedure termination
 - ⇒ `in out`: may be read and modified; is initially defined by the actual parameter and will be assigned to the actual parameter upon procedure termination
- ❑ Functions can only have in-parameters
- ❑ in-out parameters are implemented by copy-in/copy-out for elementary types
- ❑ in-out parameters are implemented either by copy-in/copy-out or by reference for composite types

Subprogram Specification

Subprogram specifications

- ❑ determines how the subprogram has to be called

Examples

```
procedure Count_Leaves_On_Binary_Tree;
```

```
procedure Push (Element: in Integer;  
                On: in out Buffer);
```

```
function "*" (X, Y: in Matrix) return Matrix;
```

- ❑ `in` is default, may also be omitted
- ❑ general rule: don't use too many parameters

Subprogram Bodies

Bodies

- ❑ subprogram specification must be completed by a corresponding body (there are certain exceptions, however, for imported subprograms written in a different language)
- ❑ body contains local declarations and a sequence of statements between **begin** and **end**

Example

```
procedure Push    (Element: in Integer;  
                  On: in out Buffer) is  
  
begin  
    On.Index := On.Index+1;  
    On.Value(On.Index) := Element;  
end Push;
```

Subprogram Calls

There are three ways to pass the actual parameter in a subprogram call:

- (1) positional notation
- (2) named parameter association
- (3) use of default parameters

Positional notations and named parameter association can be mixed - where named parameters must be listed after positional arguments.

Subprogram Calls

Positional notations

```
procedure Search_File(Key: in Name;  
                      Index: out File_Index);  
  
procedure Sleep(Time: in Duration := 10.0);  
  
procedure Sort      (Data: in out Names;  
                    Order: in Direction := Ascending);  
  
Search_File("Smith, J", Record_Entry);  
  
Sleep(120.0);  
  
Sort(Personnel_Names, Descending);
```

Subprogram Calls

Named parameter associaton

```
procedure Search_File(Key: in Name;
                    Index: out File_Index);

procedure Sleep(Time: in Duration := 10.0);

procedure Sort    (Data: in out Names;
                  Order: in Direction := Ascending);

Search_File (Index => Record_Entry,
            Key => "Smith, J");

Sleep(Time => 120.0);

Sort(Data => Personnel_Names, Order => Descending);
```

Subprogram Calls

Use of default parameters

```
procedure Search_File(Key: in Name;  
                    Index: out File_Index);  
  
procedure Sleep(Time: in Duration := 10.0);  
  
procedure Sort      (Data: in out Names;  
                    Order: in Direction := Ascending);  
  
Sleep;  
  
Sort(Personnel_Names);
```

Functions

particularities of functions (1)

- ❑ signature: identifier after `return` specifies result type
- ❑ body: expression after `return` defines result

```
function Largest_Dollar(List: in Dollarlist)
    return Dollar is

    Largest_So_Far: Dollar := Dollar'First;
begin
    ... -- computation
    return Largest_So_Far;
end Largest_Dollar;
```

Functions

particularities of functions (2)

- ❑ call to a function only within an expression context

```
Largest_Dollar (Deposits) < 1000.00
```

- ❑ function names may be predefined operators (like "+", "*", "-")

```
function "+" (Left, Right: in Integer) return Integer;
```

```
I := "+"(I, 1); -- call to "+"
```

```
I := I + 1; -- infix notation
```

Overloading

Motivation

- for a uniform programming style, subprograms may have the same name:
 - ⇒ Put_Integer (I : Integer); Put_Float (F : Float);
 - ⇒ Put (I : Integer); Put (F : Float);

Example:

```
function "+" (Left, Right: in Integer)
    return Integer;

function "+" (Left, Right: in Complex)
    return Complex;
```

Overloading in Ada 95

Consequences

- ❑ the same name may have different meanings
- ⇒ subprograms may be overloaded

Prerequisites

- ❑ no two subprograms may have the same signature
- ❑ signature is determined by
 - ⇒ name of the subprogram
 - ⇒ base type, number, and order of formal parameters
 - ⇒ result type in the case of functions

Overloading in Ada 95

Generalization

- ❑ beside of operators and subprograms, tasks and identifiers of enumeration types may be overloaded

Examples

```
procedure Order (I, J: in out Integer);
```

```
procedure Order (I, J: in out Float);
```

```
N1, N2: Positive;
```

```
...
```

```
Order(N1, N2); -- Integer Order is called
```

Overloading in Ada 95

Resolution of Overloading

```
type Color is (Red, Green, Blue);  
type Light is (Red, Yellow, Green);  
  
procedure Put (Element: in Color):  
procedure Put (Next_Phase: in Light);  
  
Put(Red); -- illegal  
Put(Blue); -- legal  
Put(Color'(Red)); -- legal  
Put(Next_Phase => Red); -- legal
```

Overloading in Ada 95

Restriction

Objects and types cannot be overloaded.

Examples

```
I : Integer;           J: Float;           type T is ...  
I : Float;            type J is ...         type T is ...
```

are not allowed in the same scope.

Statements

Assignment

```
A_Variable_1 := Some_Value;
```

- names of variables start with a letter and may contain (any number) of letters and digits. Underscores may be used but not at the end and not in sequence.
- Ada is not case-sensitive

Statements in Ada 95

If Statement

```
if A_Boolean_Expression then  
    A_Statement;  
    Another_Statement;  
end if;
```

```
if A_Boolean_Expression then  
    A_Sequence_of_Statements;  
else  
    An_Alternative_Sequence_of_Statements;  
end if;
```

Statements in Ada 95

If Statement: the elsif

```
if B1 then
    S1;
elsif B2 then
    S2;
else
    S3;
end if;
```

```
if B1 then
    S1;
else
    if B2 then
        S2;
    else
        S3;
    end if;
end if;
```

- ❑ both alternatives are equivalent
- ❑ but the *elsif* makes your program more readable

Statements in Ada 95

Case Statement

```
case An_Integer_Expression is  
  when 1      => Statements_for_Value_1;  
  when 2..6   => Statements_for_Values_2_thru_6;  
  when 8|11   => Statements_for_Values_8_or_11;  
  when others => Statements_for_all_other_Values;  
end case;
```

- ❑ case selector is `An_Integer_Expression`, may be any discrete type
- ❑ ranges are denoted by "left..right", alternatives are separated by "|"
- ❑ a *when clause* must be provided for each possible value of the case selector (you may use *others*, however)

Loop and Exit Statements (1)

loop

 Some_Statements;

exit when A_Boolean_Condition;

end loop;

A_Named_Loop:

loop

 Some_Statements;

exit A_Named_Loop **when** A_Boolean_Condition;

end loop A_Named_Loop;

- ❑ **exit** may be used everywhere in the loop body
 - ⇒ you should prefer conditional loops (see below)
- ❑ **exit** terminates the outer loop
 - ⇒ use **exit A_Named_Loop** to terminate named loop (several nesting levels)

Loop and Exit Statements (2)

```
while A_Boolean_Expression loop  
    Some_Statements;  
end loop;
```

```
for Index in 0..10 loop -- Index is implicitly declared!  
    Some_Statements;  
end loop;
```

```
A_Backward_Loop:  
    for Index in reverse 0..10 loop  
        Some_Statements;  
    end loop A_Backward_Loop;
```

Block Statement

```
A_Block_Statement:  
  declare  
    Found: Boolean := False;  
    Count: Integer := 0;  
  begin  
    while not Found loop  
      Count := Count + 1;  
      Found := Look_For_It;  
    end loop;  
  end A_Block_Statement;
```

- blocks are used to group related statements (and are used to catch exceptions)
- declared variables have local scope
- block name is optional

operators and precedence

list of operators

```
**  not  abs
*   /   mod  rem
+   -
+   -   &
=   /=   <   <=   >   >=
and  or   xor
```

↑ highest
precedence
↓ lowest

additional operators

```
in   not in
and then   or else
```

Expressions

examples

`B**2`

`Line_Count mod Page_Size`

`N not in 1..10`

`abs(-3)`

`if Y/=0 and then X/Y > 0 then ...`

`Index=0 or else Item_Found`

Types

most important requirement: type safety

- ❑ most errors should be detected at compile time
 - ⇒ *strict* type model
- ❑ type conversions are possible but must be explicit rather than implicit
- ❑ name equivalence rather than structural equivalence

Types in Ada

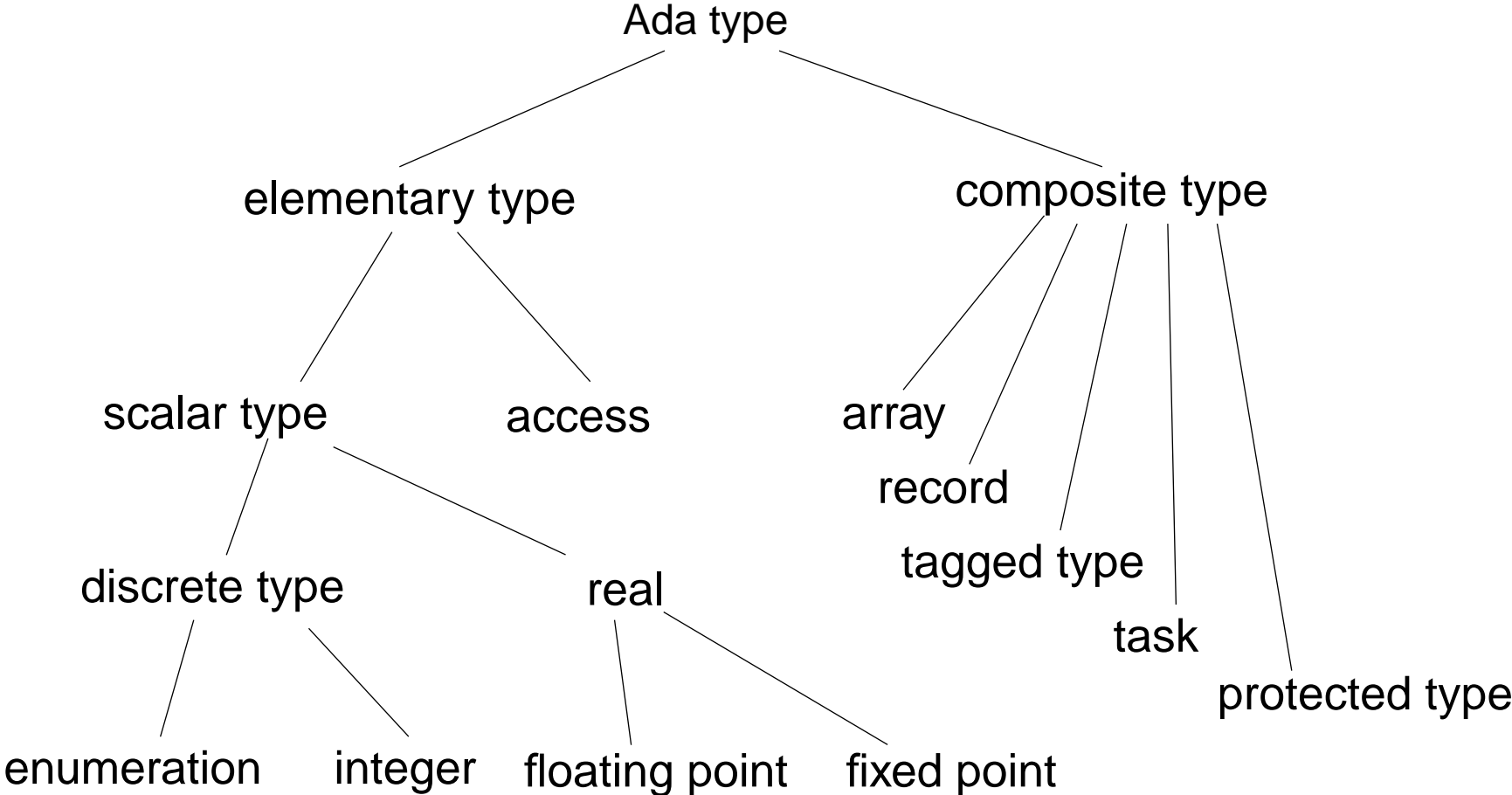
further requirement: flexibility

- ❑ rich set of predefined types
- ❑ manifold means to express user-defined types such as inheritance, subtyping, parameterization,...

third requirement: efficiency

- ❑ due to the goal of supporting real-time systems
- ❑ type model must be accordingly
- ❑ often concessions for compiler vendors

classes of types



properties of classes of types

Scalar Types

- ❑ relational operators, such as “=”, “<”, “>”, etc., are defined
- ❑ successor (Succ) and predecessor function (Pred) defined

Discrete Types

- ❑ position of a value within the type declaration: T'Pos (x)
- ❑ value at a given position: T'Val (5)

Real Types

- ❑ precision can be defined
- ❑ floating point as well as fixed point arithmetic

Types and Subtypes

a type is...

- ❑ as set of values
- ❑ and a set of corresponding (*primitive*) operations

a subtype is...

- ❑ the combination of
 - ⇒ an (existing) type and
 - ⇒ restrictions for its set of values
- ❑ no type of its own
 - ⇒ variables of different subtypes can be assigned to each other (there is a runtime check, however)

Ranges

Ranges

- ❑ Used to define a range of values
- ❑ Example:

```
-10 .. 10  
X .. X + 1  
0.0 .. 2.0 * Pi  
1 .. 0 -- null range
```

Range Constraints

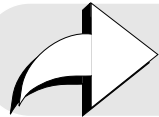
- ❑ Used to define subtypes
- ❑ Prefixing keyword **range**

```
range 1..100
```

object declarations

create two objects
with views `v1` and `v2`

views are
immutable



```
v1,v2: [aliased] [constant] type [:= expression];
```

several views to the
objects are possible

initial value for newly
created objects

Examples

```
Light_Year: constant Integer := 5_878_000_000_000;
```

```
Binary_Base: constant := 2#1111_1111#;
```

```
Octal_Base: Integer := 8#377#;
```

object declarations

Aliased

- ❑ use of 'aliased' enables copying the *reference* of a variable

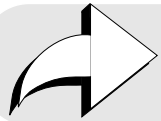
```
V1: aliased T;
```

```
type TA is access all T;
```

```
V2: TA := V1'Access;
```

- ❑ a change via V2.all will affect the value of V1 and vice versa
- ❑ 'Access is applicable only to those variables that are marked aliased
- ❑ programmer has explicit control of aliasing effects

type declarations



`type id is ...`

- ❑ every type declaration introduces a new type and - at the same time - its first subtype (well, just to simplify the language reference manual)

- ❑ a type introduced in an object declaration is called *anonymous* and does not have a subtype, e.g.,

```
V: array(...) of Integer;
```

- ❑ components of composite types require a *subtype*

```
type R is record
```

```
  X: array(...) of Integer; -- illegal!
```

```
end record;
```

- ❑ you better avoid anonymous types

enumeration types

- ❑ enumeration defines an order for its values
- ❑ first value has position 0, second has position 1, and so forth
- ❑ positional access via type attributes (see below)
- ❑ literals of the enumeration are implicitly defined parameter-less functions

```
type Day is (Mon, Tue, Wed, Thu, Fre, Sat, Sun);
```

```
type Latin_Digit is ('I', 'V', 'X', 'L', 'C', 'D', 'M');
```

```
type Color is (Red, Yellow, Green, Blue, Brown);
```

```
subtype Working_Day is Day range Mon..Fre;
```

```
subtype Rainbow is Color range Red..Blue;
```

enumeration types (2)

Boolean

```
type Boolean is (False, True);  
function "and"(Left,Right: Boolean) return Boolean  
-- "or", "xor", "not" analogously
```

Character

```
type Character is (nul, ..., '0', '1', ..., 'A', ...);  
-- 255 characters  
type Wide_Character is (nul, ..., FFFF);  
-- ISO 10646 BMP character set: 65536 (2**16) characters
```



predefined types in Ada

Integer Types

„Signed“ Integer Types

```
type Integer is range implementation-defined;

function "+"(Left,Right: Integer) return Integer;
-- dto. für "-", "*", "/", "**", "rem", "mod"

function "+"(Right: Integer) return Integer;
-- dto. für "-", "abs"

subtype Natural is Integer range 0..Integer'Last;
subtype Positive is Integer range 1..Integer'Last;
```



predefined types in Ada

Integer Types (2)

Modular Integer Types

- ❑ unsigned integer types

- ❑ Modulo-arithmetic, e.g.,

```
type Hash_Index is mod 97;
```

```
type Unsigned_Byte is mod 256; -- 0..255
```

...

```
u: Unsigned_Byte;
```

```
u := 128 + 128; -- OK: u=0 (256 mod 256)
```

- ❑ "and", "or", "xor", "not" for modular types are defined as bitwise combination of the values

Real Types

Floating Point Types

- specification of relative precision via number of relevant decimals (mantissa)
 - ⇒ high precision around 0
 - ⇒ low precision for large numbers

- Examples:

```
type Real is digits 8;
```

```
type Probability is digits 6 range -1.0..1.0;
```

- Predefined type:

```
type Float is digits implementation-defined;
```

```
function "*" (L:Float; R:Integer) return Float;  
-- dto. für "+", "-", "*", "/"
```

Real Types (2)

Fixed Point Types

- specification of an absolute precision

```
type Volt is delta 0.125 range 0.0..255.0;
```

(N.B.: the right border is not included!)

- optional: number of relevant positions where Delta is a decimal power

```
type Euro is delta 0.01 digits 15;
```

```
subtype Salary is Euro digits 10;
```

! Euro'Last = 10.0^{**13} - 0.01 !

▶

```
type Percent is delta 0.01 range 0.0 .. 1.01;
```

Type attributes

What is an attribute?

- Access to type characteristics
- Syntax: `Typ'Attribute_Name`
- Allows more flexibility and improves portability

Attribute of scalar types

- ❑ First/Last: upper/lower limit of the range of values
- ❑ Pred/Succ: predecessor/successors for ranges of values
- ❑ Image/Value: conversion into/from string representation for input/output

```
subtype Positive is Integer range 1..Integer'Last;  
P: Positive;  
  
...  
  
if Positive'Succ(P) > 1  
  then Put(Positive'Image(P));  
  else Put("Compiler error!");  
end if;
```

Arrays

Constrained Arrays

- ❑ size is determined at elaboration time (evaluation of type declarations at runtime)
- ❑ size cannot be changed anymore

```
type Table is array (1..10) of Integer;
```

```
type Schedule is array (Day) of Boolean;
```

```
type Matrix is array(1..100, 1..100) of Float;
```

```
type MyArray is array (F(X)..G(Y)) of Character;
```

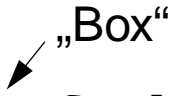
```
type Working_Schedule is array(Day range Mon..Fre)  
      of Boolean;
```

Arrays (2)

Unconstrained Arrays

- ❑ size is determined at *object creation*
- ❑ size cannot be changed either (!)

```
type Vector is array (Positive range <>) of Float;
type Flexible_Matrix is array
  (Integer range <>, Integer range <>) of Float;
...
V: Vector(1..5);
M: Flexible_Matrix(1..100,1..100);
procedure F(V: Vektor); -- !!
```



Array Attributes

```
A: Flexible_Matrix(1..100, 2..200);
```

- lower bound: `A'First = 1`
- upper bound: `A'Last = 100`
- number of elements: `A'Length = 100`
- index range: `A'Range = 1..100`

All attributes can also be used with a parameter:

```
'First(N), 'Last(N), 'Length(N), 'Range(N)
```

N denotes the N'th dimension of the array

Example: Stack

```
type Stack is array (Positive range <>) of Elem;  
function push(S:Stack; El:Elem) return Stack;  
function pop(S:Stack) return Stack;  
function top(S:Stack) return Elem;  
  
...  
S: Stack(1..100);  
E: Elem := ...;  
S := Push(S,E);  
S := Pop(S);  
E := Top(S);
```

Strings

String (predefined)

```
type String is
  array(Positive range <>) of Character;

function "&"(L:String; R:String) return String;
function "&"(L:String; R:Character) return String;
function "&"(L:Character; R:String) return String;
function "&"(L,R:Character) return String;

function "="(L,R:String) return Boolean;
-- dto. für "<", "<=", etc.
```

Wide_String (predefined)

```
type Wide_String is
  array(Positive range <>) of Wide_Character;
```

Strings (2)

Examples...

```
Stars: String(1..120) := (1..120 => '*');
```

```
Question: String := "How many characters?";
```

```
Ask_Twice: String := Question & Question;
```

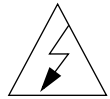
Comparison

```
"Pi" < "Piste"    -- True
```

```
"ada?" < "Ada!"  -- False: Ascii is relevant
```

Strings (3)

Be cautious with assignments!



```
Ask_Twice := Question; -- Runtime error!  
Ask_Twice(1..Question'Length) := Question; -- OK!  
Ask_Twice (N..M) := Question; -- Runtime check  
Question := Ask_Twice (1..Question'Length); -- OK!
```

Records

```
type Transaction is record
  Account: Account_Number;
  Password: Password_Type;
  Amount: DM := 0.0;
end record;

...
Last_Transaction: Transaction;

...
Last_Transaction.Account := 55_238_1234;
Last_Transaction.Amount := 1250.00;
Last_Transaction.Password := "Guess_What";
```

Record Aggregate

Positional

```
Transaction1 := (55_238_1245, "XRC7p_d", 125.00);
```

Named

```
Transaction2 :=  
  (Amount    => 125.00,  
   Account   => 55_238_1245,  
   Password  => "XRC7p_d");
```

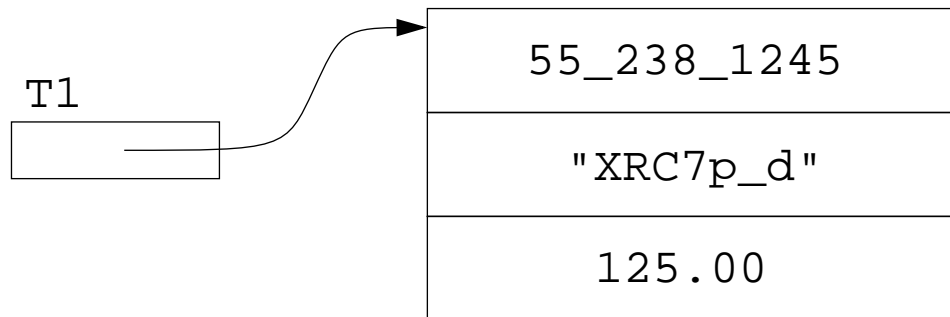
| |
|-------------|
| 55_238_1245 |
| "XRC7p_d" |
| 125.00 |

Access Types

- ❑ „Pointer“-Types
- ❑ Dynamic objects created on the heap

```
type Transactions_Name is access Transaction;
```

```
T1: Transactions_Name := new Transaction;  
                        -- Create transaction object
```



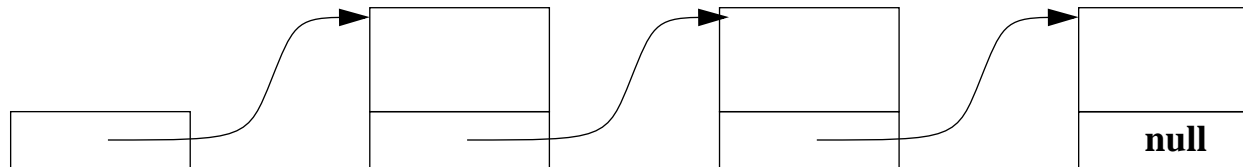
```
T1.all.Amount := 125.00;    -- explicit dereference
```

```
T1.all := Last_Transaction;
```

```
T1.Amount := 125.00;      -- implicit dereference
```

Access Typen (2)

```
type Transaction; -- incomplete declaration  
type Transaction_Name is access Transaction;  
type Transaction is record  
  Account : Account_Number;  
  Password: Password_Type;  
  Amount  : DM := 0.0;  
  Next_Transaction: Transaction_Name  
end record;
```



Access Types (3)

Deallocation


```
with Unchecked_Deallocation;
procedure Main is
  type T is ...;
  type PT is access T;
  procedure Dispose is
    new Unchecked_Deallocation ( Object => T,
                                Name   => PT );

  my_pt : PT;
begin
  Dispose (my_pt);
end;
```


Type Equivalence

When is an assignment legal?

```
type A is array(1..10) of Boolean;
type B is array(1..10) of Boolean;

A1: A;
B1: B;
C1,C2: array(1..10) of Boolean;
...
A1 := B1;
C1 := C2;  illegal
```

Rule

 Every application of a type constructor creates a new type!

Type Conversions

- ❑ Strict name equivalence in Ada is sometimes too restrictive
- ❑ Value or its representation, respectively, is “transformed” via conversions (possibly with a constraint check at runtime)
- ❑ Type conversions are defined
 - ⇒ positionally for enumerations
 - ⇒ recursively for composite types
 - ⇒ by means of the target type for access types
- ❑ `Unchecked_Conversion` is *always* possible
 - ⇒ re-interpretation of the bits
 - ⇒ useful for systems programming
 - ⇒ be cautious!

Type Conversions (2)

Examples

```
Real(2*5)      -- Value is 10.0 (Floating Point)
Integer(1.6)   -- Value is 2

type Sequence is array (Int range <>) of Int;
subtype Dozen is Sequence (1..12);
List: array (1..20) of Int;
...
Sequence(List)           -- bounds of List
Sequence(List(11..20))  -- bounds 11 und 20
Dozen(List(9..20))      -- bounds of Dozen
```

Type Representation Clauses

Specification of memory layout

- ❑ Eases (allows, respectively) communication with system written in different languages
- ❑ Record components

```
type My_Rec is record A,B: Integer; end record;
```

```
for My_Rec use record  
  A at 0 range 0..31;  
  B at 4 range 0..31;  
end record;
```

- ❑ Order of enumeration types

```
type My_Enum is (One,Two,Three);
```

```
for My_Enum use (One => 1, Two => 2, Three => 3);
```

Type Representation Clauses (2)

Representation attributes

- ❑ S'Size = Size in Bits
- ❑ S'Address = Address of the first memory element
- ❑ S.C'Position = S.C'Address - S'Address
- ❑ S.C'First_Bit = offset of the first bit of C in bits
- ❑ S.C'Last_Bit = Offset of the last bit of C in bits

```
for My_Rec'Size use 2*System.Storage_Unit;
```