Rationale for Ada 2005: 6 Predefined library

John Barnes

John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This paper describes various improvements to the predefined library in Ada 2005.

There are a number of important new core packages in Ada 2005. These include a number of packages for the manipulation of various types of containers, packages for directory operations and packages providing access to environment variables.

The entire ISO/IEC 10646:2003 character repertoire is now supported. Program text may now include other alphabets (such as Cyrillic and Greek) and wide-wide characters and strings are supported at run-time. There are also some improvements to the existing character, string and text input–output packages.

The Numerics annex now includes vector and matrix operations including those previously found in the secondary standard ISO/IEC 13813.

This is one of a number of papers concerning Ada 2005 which are being published in the Ada User Journal. An earlier version of this paper appeared in the Ada User Journal, Vol. 26, Number 4, December 2005. Other papers in this series will be found in later issues of the Journal or elsewhere on this website.

Keywords: rationale, Ada 2005.

1 Overview of changes

The WG9 guidance document [1] says

"The main purpose of the Amendment is to address identified problems in Ada that are interfering with Ada's usage or adoption, especially in its major application areas (such as high-reliability, long-lived real-time and/or embedded applications and very large complex systems). The resulting changes may range from relatively minor, to more substantial."

Certainly one of the stated advantages of languages such as Java is that they come with a huge predefined library. By contrast the Ada library is somewhat Spartan and extensions to it should make Ada more accessible.

The guidance document also warns about secondary standards. Its essence is don't use secondary standards if you can get the material into the RM itself. And please put the stuff on vectors and matrices from ISO/IEC 13813 [2] into the RM. The reason for this exhortation is that secondary standards have proved themselves to be almost invisible and hence virtually useless.

We have already discussed the additional library packages in the area of tasking and real time in a previous paper. The following Ada issues cover the relevant changes in other areas and are described in detail in this paper:

161 Preelaborable initialization

248 Directory operations

270 Stream item size control

273 Use of PCS should not be normative

285 Support for 16-bit and 32-bit characters

296 Vector and matrix operations

301 Operations on language-defined strings

302 Container library

328 Non-generic version of Complex_IO

351 Time operations

362 Some predefined packages should be recategorized

366 More liberal rules for Pure units

370 Add standard interface for environment variables

388 Add Greek pi to Ada.Numerics

395 Clarifications concerning 16- and 32-bit characters

400 Wide and wide-wide images

418 Vector norm

427 Default parameters and Calendar operations

428 Input–output for bounded strings

441 Null streams

These changes can be grouped as follows.

First the container library is rather extensive and merits a whole paper alone (302). We only refer to it here for completeness.

New child packages of Calendar provide extra facilities for manipulating times and dates (351, 427).

There are additional packages in the core library providing access to aspects of the operational environment. These concern directory operations (248) and environment variables (370).

There are changes concerning characters both for writing program text itself and for handling characters and strings at run time. There is now support for 16- and 32-bit characters (285, 388, 395, 400), and there are additional operations in the string packages (301, 428).

The Numerics annex is enhanced by the addition of the vector and matrix material previously in ISO/IEC 13813 plus some commonly required linear algebra algorithms (296, 418) and a trivial addition concerning complex input–output (328).

The categorization of various predefined units has been changed in order to remove unnecessary restrictions on their use in distributed systems and similar applications (362, 366). The new pragma Preelaborable_Initialization is introduced as well for similar reasons (161). We can also group a minor change to the Distributed Systems annex here (273).

Finally there is new attribute Stream_Size in order to increase the portability of streams (270) and the parameter Stream of Read, Write etc now has a null exclusion (441).

2 The container library

This is a huge addition to the language and is described in a separate paper for convenience.

3 Times and dates

The first change to note is that the subtype Year_Number in the package Ada.Calendar in Ada 2005 is

```
subtype Year_Number is Integer range 1901 .. 2399;
```

In Ada 95 (and in Ada 83) the range is 1901 .. 2099. This avoids the leap year complexity caused by the 400 year rule at the expense of the use of dates in the far future. But, the end of the 21st century is perhaps not so far into the future, so it was decided that the 2.1k problem should be solved now rather than later. However, it was decided not to change the lower bound because some systems are known to have used that as a time datum. The upper bound was chosen in order to avoid difficulties for implementations. For example, with one nanosecond for Duration'Small, the type Time can just be squeezed into 64 bits.

Having grasped the nettle of doing leap years properly Ada 2005 dives in and deals with leap seconds, time zones and other such matters in pitiless detail.

There are three new child packages Calendar.Time_Zones, Calendar.Arithmetic and Calendar.Formatting. We will look at these in turn.

The specification of the first is

```
package Ada.Calendar.Time_Zones is

  -- Time zone manipulation:
  type Time_Offset is range –28*60 .. 28*60;
  Unknown_Zone_Error: exception;

  function UTC_Time_Offset(Date: Time := Clock) return Time_Offset;
end Ada.Calendar.Time_Zones;
```

Time zones are described in terms of the number of minutes different from UTC (which curiously is short for Coordinated Universal Time); this is close to but not quite the same as Greenwich Mean Time (GMT) and similarly does not suffer from leaping about in spring and falling about in the autumn. It might have seemed more natural to use hours but some places (for example India) have time zones which are not an integral number of hours different from UTC.

Time is an extraordinarily complex subject. The difference between GMT and UTC is never more than one second but at the moment of writing there is a difference of about 0.577 seconds. The BBC broadcast timesignals based on UTC but call them GMT and with digital broadcasting they turn up late anyway. The chronophile might find the website www.merlyn.demon.co.uk/misctime.htm#GMT of interest.

So the function UTC_Time_Offset applied in an Ada program in Paris to a value of type Time in summer should return a time offset of 120 (one hour for European Central Time plus one hour for daylight saving or heure d'été). Remember that the type Calendar.Time incorporates the date. To find the offset now (that is, at the time of the function call) we simply write

```
Offset := UTC_Time_Offset;
```

and then Clock is called by default.

To find what the offset was on Christmas Day 2000 we write

```
Offset := UTC_Time_Offset(Time_Of(2000, 12, 25));
```

and this should return 60 in Paris. So the poor function has to remember the whole history of local time changes since 1901 and predict them forward to 2399 – these Ada systems are pretty smart! In reality the intent is to use whatever the underlying operating system provides. If the information is not known then it can raise Unknown_Zone_Error.

Note that we are assuming that the package Calendar is set to the local civil (or wall clock) time. It doesn't have to be but one expects that to be the normal situation. Of course it is possible for an Ada system running in California to have Calendar set to the local time in New Zealand but that would be unusual. Equally, Calendar doesn't have to adjust with daylight saving but we expect that it will. (No wonder that Ada.Real_Time was introduced for vital missions such as boiling an egg.)

A useful fact is that

```
Clock – Duration(UTC_Time_Offset*60)
```

gives UTC time – provided we don't do this just as daylight saving comes into effect in which case the call of Clock and that of UTC_Time_Offset might not be compatible.

More generally the type Time_Offset can be used to represent the difference between two time zones. If we want to work with the difference between New York and Paris then we could say

```
NY_Paris: Time_Offset := –360;
```

The time offset between two different places can be greater than 24 hours for two reasons. One is that the International Date Line weaves about somewhat and the other is that daylight saving time can extend the difference as well. Differences of 26 hours can easily occur and 27 hours is possible. Accordingly the range of the type Time_Offset allows for a generous 28 hours.

The package Calendar.Arithmetic provides some awkward arithmetic operations and also covers leap seconds. Its specification is

```
package Ada.Calendar.Arithmetic is

  -- Arithmetic on days:
  type Day_Count is range
    –366*(1+Year_Number'Last – Year_Number'First)
    ..
    +366*(1+Year_Number'Last – Year_Number'First);

  subtype Leap_Seconds_Count is Integer range –2047 .. 2047;

  procedure Difference(Left, Right: in Time;
                       Days: out Day_Count; Seconds: out Duration;
                       Leap_Seconds: out Leap_Seconds_Count);

  function "+" (Left: Time; Right: Day_Count) return Time;
  function "+" (Left: Day_Count; Right: Time) return Time;
  function "–" (Left: Time; Right: Day_Count) return Time;
  function "–" (Left, Right: Time) return Day_Count;

end Ada.Calendar.Arithmetic;
```

The range for Leap_Seconds_Count is generous. It allows for a leap second at least four time a year for the foreseeable future – the somewhat arbitrary range chosen allows the value to be accommodated in 12 bits. And the 366 in Day_Count is also a bit generous – but the true expression would be very unpleasant.

One of the problems with the old planet is that it is slowing down and a day as measured by the Earth's rotation is now a bit longer than 86,400 seconds. Naturally enough we have to keep the seconds uniform and so in order to keep worldly clocks synchronized with the natural day, an odd leap second has to be added from time to time. This is always added at midnight UTC (which means it can pop up in the middle of the day in other time zones). The existence of leap seconds makes calculations with times somewhat tricky.

The basic trouble is that we want to have our cake and eat it. We want to have the invariant that a day has 86,400 seconds but unfortunately this is not always the case.

The procedure Difference operates on two values of type Time and gives the result in three parts, the number of days (an integer), the number of seconds as a Duration and the number of leap seconds (an integer). If Left is later then Right then all three numbers will be nonnegative; if earlier, then nonpositive.

Remember that Difference like all these other operations always works on local time as defined by the clock in Calendar (unless stated otherwise).

Suppose we wanted to find the difference between noon on June 1st 1982 and 2pm on July 1st 1985 according to a system set to UTC. We might write

```
Days: Day_Count;
Secs: Duration;
Leaps: Leap_Seconds_Count;
...
Difference(Time_Of(1985, 7, 1, 14*3600.0),
           Time_Of(1982, 6, 1, 12*3600.0), Days, Secs, Leaps);
```

The results should be

```
Days = 365+366+365+30 = 1126,
Secs = 7200.0,
Leaps = 2.
```

There were leap seconds on 30 June 1983 and 30 June 1985.

The functions "+" and "–" apply to values of type Time and Day_Count (whereas those in the parent Calendar apply only to Time and Duration and thus only work for intervals of a day or so). Note that the function "–" between two values of type Time in this child package produces the same value for the number of days as the corresponding call of the function Difference – leap seconds are completely ignored. Leap seconds are in fact ignored in all the operations "+" and "–" in the child package.

However, it should be noted that Calendar."–" counts the true seconds and so the expression

```
Calendar."–" (Time_Of(1985, 7, 1, 1*3600.0), Time_Of(1985, 6, 30, 23*3600.0))
```

has the Duration value 7201.0 and not 7200.0 because of the leap second at midnight that night. (We are assuming that our Ada system is running at UTC.) The same calculation in New York will produce 7200.0 because the leap second doesn't occur until 4 am in EST (with daylight saving).

Note also that

```
Calendar."–" (Time_Of(1985, 7, 1, 0.0), Time_Of(1985, 6, 30, 0.0))
```

in Paris where the leap second occurs at 10pm returns 86401.0 whereas the same calculation in New York will return 86400.0.

The third child package Calendar.Formatting has a variety of functions. Its specification is

```
with Ada.Calendar.Time_Zones;
use Ada.Calendar.Time_Zones;
package Ada.Calendar.Formatting is
```

```
-- Day of the week:
type Day_Name is (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);

function Day_Of_Week(Date: Time) return Day_Name;

-- Hours:Minutes:Seconds access:
subtype Hour_Number is Natural range 0 .. 23;
subtype Minute_Number is Natural range 0 .. 59;
subtype Second_Number is Natural range 0 .. 59;
subtype Second_Duration is Day_Duration range 0.0 .. 1.0;

function Year(Date: Time; Time_Zone: Time_Offset := 0) return Year_Number;

-- similarly functions Month, Day, Hour, Minute

function Second(Date: Time) return Second_Number;

function Sub_Second(Date: Time) return Second_Duration;

function Seconds_Of(Hour: Hour_Number;
          Minute: Minute_Number;
          Second: Second_Number := 0;
          Sub_Second: Second_Duration := 0.0) return Day_Duration;

procedure Split(Seconds: in Day_Duration;        -- (1)
          Hour: out Hour_Number;
          Minute: out Minute_Number;
          Second: out Second_Number;
          Sub_Second: out Second_Duration);

procedure Split(Date: in Time;                   -- (2)
          Year: out Year_Number;
          Month: out Month_Number;
          Day: out Day_Number;
          Hour: out Hour_Number;
          Minute: out Minute_Number;
          Second: out Second_Number;
          Sub_Second: out Second_Duration;
          Time_Zone: in Time_Offset := 0);

function Time_Of(Year: Year_Number;
          Month: Month_Number;
          Day: Day_Number;
          Hour: Hour_Number;
          Minute: Minute_Number;
          Second: Second_Number;
          Sub_Second: Second_Duration := 0.0;
          Leap_Second: Boolean := False;
          Time_Zone: Time_Offset := 0) return Time;

function Time_Of(Year: Year_Number;
          Month: Month_Number;
          Day: Day_Number;
          Seconds: Day_Duration;
          Leap_Second: Boolean := False;
          Time_Zone: Time_Offset := 0) return Time;
```

```
procedure Split(Date: in Time;                         -- (3)
          ... -- as (2) but with additional parameter
          Leap_Second: out Boolean;
          Time_Zone: in Time_Offset := 0);

procedure Split(Date: in Time;                         -- (4)
          ... -- as Calendar.Split
          ... -- but with additional parameters
          Leap_Second: out Boolean;
          Time_Zone: in Time_Offset := 0);

-- Simple image and value:
function Image(Date: Time;
          Include_Time_Fraction: Boolean := False;
          Time_Zone: Time_Offset := 0) return String;

function Value(Date: String; Time_Zone: Time_Offset := 0) return Time;

function Image (Elapsed_Time: Duration;
          Include_Time_Fraction: Boolean := False) return String;

function Value(Elapsed_Time: String) return Duration;

end Ada.Calendar.Formatting;
```

The function Day_Of_Week will be much appreciated. It is a nasty calculation.

Then there are functions Year, Month, Day, Hour, Minute, Second and Sub_Second which return the corresponding parts of a Time taking account of the time zone given as necessary. It is unfortunate that functions returning the parts of a time (as opposed to the parts of a date) were not provided in Calendar originally. All that Calendar provides is Seconds which gives the number of seconds from midnight and leaves users to chop it up for themselves. Note that Calendar.Second returns a Duration whereas the function in this child package is Seconds which returns an Integer. The fraction of a second is returned by Sub_Second.

Most of these functions have an optional parameter which is a time zone offset. Wherever in the world we are running, if we want to know the hour according to UTC then we write

```
Hour(Clock, UTC_Time_Offset)
```

If we are in New York and want to know the hour in Paris then we write

```
Hour(Clock, –360)
```

since New York is 6 hours (360 minutes) behind Paris.

Note that Second and Sub_Second do not have the optional Time_Offset parameter because offsets are an integral number of minutes and so the number of seconds does not depend upon the time zone.

The package also generously provides four procedures Split and two procedures Time_Of. These have the same general purpose as those in Calendar. There is also a function Seconds_Of. We will consider them in the order of declaration in the package specification above.

The function Seconds_Of creates a value of type Duration from components Hour, Minute, Second and Sub_Second. Note that we can use this together with Calendar.Time_Of to create a value of type Time. For example

```
T := Time_Of(2005, 4, 2, Seconds_Of(22, 4, 10, 0.5));
```

makes the time of the instant when I (originally) typed that last semicolon.

The first procedure Split is the reverse of Seconds_Of. It decomposes a value of type Duration into Hour, Minute, Second and Sub_Second. It is useful with the function Calendar.Split thus

```
Split(Some_Time, Y, M, D, Secs);  -- split time
Split(Secs, H, M, S, SS);          -- split secs
```

The next procedure Split (no 2) takes a Time and a Time_Offset (optional) and decomposes the time into its seven components. Note that the optional parameter is last for convenience. The normal rule for parameters of predefined procedures is that parameters of mode in are first and parameters of mode out are last. But this is a nuisance if parameters of mode in have defaults since this forces named notation if the default is used.

There are then two functions Time_Of which compose a Time from its various constituents and the Time_Offset (optional). One takes seven components (with individual Hour, Minute etc) whereas the other takes just four components (with Seconds in the whole day). An interesting feature of these two functions is that they also have a Boolean parameter Leap_Second which by default is False.

The purpose of this parameter needs to be understood carefully. Making up a typical time will have this parameter as False. But suppose we need to compose the time midway through the leap second that occurred on 30 June 1985 and assign it to a variable Magic_Moment. We will assume that our Calendar is in New York and set to EST with daylight saving (and so midnight UTC is 8pm in New York). We would write

```
Magic_Moment: Time := Time_Of(1985, 6, 30, 19, 59, 59, 0.5, True);
```

In a sense there were two 19:59:59 that day in New York. The proper one and then the leap one; the parameter distinguishes them. So the moment one second earlier is given by

```
Normal_Moment: Time := Time_Of(1985, 6, 30, 19, 59, 59, 0.5, False);
```

We could have followed ISO and used 23:59:60 UTC and so have subtype Second_Number **is** Natural **range** 0 .. 60; but this would have produced an incompatibility with Ada 95.

Note that if the parameter Leap_Second is True and the other parameters do not identify a time of a leap second then Time_Error is raised.

There are then two corresponding procedures Split (nos 3 and 4) with an out parameter Leap_Second. One produces seven components and the other just four. The difference between this seven-component procedure Split (no 3) and the earlier Split (no 2) is that this one has the out parameter Leap_Second whereas the other does not. Writing

```
Split(Magic_Moment, 0, Y, M, D, H, M, S, SS, Leap);
```

results in Leap being True whereas

```
Split(Normal_Moment, 0, Y, M, D, H, M, S, SS, Leap);
```

results in Leap being False but gives all the other out parameters (Y, ... , SS) exactly the same values.

On the other hand calling the version of Split (no 2) without the parameter Leap_Second thus

```
Split(Magic_Moment, 0, Y, M, D, H, M, S, SS);
Split(Normal_Moment, 0, Y, M, D, H, M, S, SS);
```

produces exactly the same results.

The reader might wonder why there are two Splits on Time with Leap_Second but only one without. This is because the parent package Calendar already has the other one (although without the time zone parameter). Another point is that in the case of Time_Of, we can default the Leap parameter being of mode in but in the case of Split the parameter has mode out and cannot be omitted. It would be bad practice to encourage the use of a dummy parameter which is ignored and hence there have to be additional versions of Split.

Finally, there are two pairs of functions Image and Value. The first pair works with values of type Time. A call of Image returns a date and time value in the standard ISO 8601 format. Thus taking the Normal_Moment above

```
Image(Normal_Moment)
```

returns the following string

```
"1985-06-30 19:59:59"          -- in New York
```

If we set the optional parameter Include_Time_Fraction to True thus

```
Image(Normal_Moment, True)
```

then we get

```
"1985-06-30 19:59:59.50"
```

There is also the usual optional Time_Zone parameter so we could produce the time in Paris (from the program in New York) thus

```
Image(Normal_Moment, True, –360)
```

giving

```
"1985-07-01 02:59:59.50"       -- in Paris
```

The matching Value function works in reverse as expected.

We would expect to get exactly the same results with Magic_Moment. However, since some implementations might have an ISO function available in their operating system it is also allowed to produce

```
"1985-06-30 19:59:60"          -- in New York
```

The other Image and Value pair work on values of type Duration thus

```
Image(10_000.0)                -- "02:46:40"
```

with the optional parameter Include_Time_Fraction as before. Again the corresponding function Value works in reverse.

4 Operational environment

Two new packages are added to Ada 2005 in order to aid communication with the operational environment. They are Ada.Environment_Variables and Ada.Directories.

The package Ada.Environment_Variables has the following specification

```
package Ada.Environment_Variables is
  pragma Preelaborate(Environment_Variables);

  function Value(Name: String) return String;
  function Exists(Name: String) return Boolean;
  procedure Set(Name: in String; Value: in String);

  procedure Clear(Name: in String);
  procedure Clear;

  procedure Iterate(Process: not null access procedure (Name, Value: in String));

end Ada.Environment_Variables;
```

This package provides access to environment variables by name. What this means and whether it is supported depends upon the implementation. But most operating systems have environment variables of some sort. And if not, the implementation is encouraged to simulate them.

The values of the variable are also implementation defined and so simply represented by strings.

The behaviour is straightforward. We might check to see if there is a variable with the name "Ada" and then read and print her value and set it to 2005 if it is not, thus

```
if not Exists("Ada") then
  raise Horror;                    -- quel dommage!
end if;

Put("Current value of Ada is ");  Put_Line(Value("Ada"));

if Value("Ada") /= "2005" then
  Put_Line("Revitalizing Ada now");
  Set("Ada", "2005");
end if;
```

The procedure Clear with a parameter deletes the variable concerned. Thus Clear("Ada") eliminates her completely so that a subsequent call Exists("Ada") will return False. Note that Set actually clears the variable concerned and then defines a new one with the given name and value. The procedure Clear without a parameter clears all variables.

We can iterate over the variables using the procedure Iterate. For example we can print out the current state by

```
procedure Print_One(Name, Value: in String) is
begin
  Put_Line(Name & "=" & Value);
end Print_One;
...
Iterate(Print_One'Access);
```

The procedure Print_One prints the name and value of the variable passed as parameters. We then pass an access to this procedure as a parameter to the procedure Iterate and Iterate then calls Print_One for each variable in turn.

Note that the slave procedure has both Name and Value as parameters. It might be thought that this was unnecessary since the user can always call the function Value. However, real operating systems can sometimes have several variables with the same name; providing two parameters ensures that the name/value pairs are correctly matched.

Attempting to misuse the environment package such as reading a variable that doesn't exist raises Constraint_Error or Program_Error.

There are big dangers of race conditions because the environment variables are really globally shared. Moreover, they might be shared with the operating system itself as well as programs written in other languages.

A particular point is that we must not call the procedures Set or Clear within a procedure passed as a parameter to Iterate.

The other environment package is Ada.Directories. Its specification is

```
with Ada.IO_Exceptions;
with Ada.Calendar;
package Ada.Directories is
```

```
-- Directory and file operations:
function Current_Directory return String;
procedure Set_Directory(Directory: in String);
procedure Create_Directory(New_Directory: in String; Form: in String := "");
procedure Delete_Directory(Directory: in String);
procedure Create_Path(New_Directory: in String; Form: in String := "");
procedure Delete_Tree(Directory: in String);
procedure Delete_File(Name: in String);
procedure Rename(Old_Name: in String; New_Name: in String);
procedure Copy_File(Source_Name: in String; Target_Name: in String; Form: in String := "");

-- File and directory name operations:
function Full_Name(Name: String) return String;
function Simple_Name(Name: String) return String;
function Containing_Directory(Name: String) return String;
function Extension(Name: String) return String;
function Base_Name(Name: String) return String;
function Compose(Containing_Directory: String := ""; Name: String; Extension: String := "")
                                                                        return String;

-- File and directory queries:
type File_Kind is (Directory, Ordinary_File, Special_File);
type File_Size is range 0 .. implementation_defined;
function Exists(Name: String) return Boolean;
function Kind(Name: String) return File_Kind;
function Size(Name: String) return File_Size;
function Modification_Time(Name: String) return Ada.Calendar.Time;

-- Directory searching:
type Directory_Entry_Type is limited private;
type Filter_Type is array (File_Kind) of Boolean;
type Search_Type is limited private;
procedure Start_Search(Search: in out Search_Type;
                       Directory: in String; Pattern: in String;
                       Filter: in Filter_Type := (others => True));
procedure End_Search(Search: in out Search_Type);
function More_Entries(Search: Search_Type) return Boolean;
procedure Get_Next_Entry(Search: in out Search_Type;
                       Directory_Entry: out Directory_Entry_Type);
procedure Search(Directory: in String;
           Pattern: in String;
           Filter: in Filter_Type := (others => True);
           Process: not null access procedure
                 (Directory_Entry: in Directory_Entry_Type));

-- Operations on Directory Entries:
function Simple_Name(Directory_Entry: Directory_Entry_Type) return String;
function Full_Name(Directory_Entry: Directory_Entry_Type) return String;
function Kind(Directory_Entry: Directory_Entry_Type) return File_Kind;
function Size(Directory_Entry: Directory_Entry_Type) return File_Size;
function Modification_Time(Directory_Entry: Directory_Entry_Type)
                                                  return Ada.Calendar.Time;
```

```
   Status_Error: exception renames Ada.IO_Exceptions.Status_Error;
   Name_Error: exception renames Ada.IO_Exceptions.Name_Error;
   Use_Error: exception renames Ada.IO_Exceptions.Use_Error;
   Device_Error: exception renames Ada.IO_Exceptions.Device_Error;
private
   -- Not specified by the language
end Ada.Directories;
```

Most operating systems have some sort of tree-structured filing system. The general idea of this package is that it allows the manipulation of file and directory names as far as is possible in a unified manner which is not too dependent on the implementation and operating system.

Files are classified as directories, special files and ordinary files. Special files are things like devices on Windows and soft links on Unix; these cannot be created or read by the predefined Ada input–output packages.

Files and directories are identified by strings in the usual way. The interpretation is implementation defined.

The full name of a file is a string such as

```
"c:\adastuff\rat\library.doc"
```

and the simple name is

```
"library.doc"
```

At least that is in good old DOS. In Windows XP it is something like

```
"C:\Documents and Settings\john.JBI3\My Documents\adastuff\rat\library.doc"
```

For the sake of illustration we will continue with the simple DOS example. The current directory is that set by the "cd" command. So assuming we have done

```
c:\>cd adastuff
c:\adastuff>
```

then the function Current_Directory will return the string "c:\adastuff". The procedure Set_Directory sets the current default directory. The procedures Create_Directory and Delete_Directory create and delete a single directory. We can either give the full name or just the part starting from the current default. Thus

```
Create_Directory("c:\adastuff\history");
Delete_Directory("history");
```

will cancel out.

The procedure Create_Path creates several nested directories as necessary. Thus starting from the situation above, if we write

```
Create_Path("c:\adastuff\books\old");
```

then it will first create a directory "books" in "c:\adastuff" and then a directory "old" in "books". On the other hand if we wrote Create_Path("c:\adastuff\rat"); then it would do nothing since the path already exists. The procedure Delete_Tree deletes a whole tree including subdirectories and files.

The procedures Delete_File, Rename and Copy_File behave as expected. Note in particular that Copy_File can be used to copy any file that could be copied using a normal input–output package such as Text_IO. For example, it is really tedious to use Text_IO to copy a file intact including all line and page terminators. It is a trivial matter using Copy_File.

Note also that the procedures Create_Directory, Create_Path and Copy_File have an optional Form parameter. Like similar parameters in the predefined input–output packages the meaning is implementation defined.

The next group of six functions, Full_Name, Simple_Name, Containing_Directory, Extension, Base_Name and Compose just manipulate strings representing file names and do not in any way interact with the actual external file system. Moreover, of these, only the behaviour of Full_Name depends upon the current directory.

The function Full_Name returns the full name of a file. Thus assuming the current directory is still "c:\adastuff"

```
Full_Name("rat\library.doc")
```

returns "c:\adastuff\rat\library.doc" and

```
Full_Name("library.doc")
```

returns "c:\adastuff\library.doc". The fact that such a file does not exist is irrelevant. We might be making up the name so that we can then create the file. If the string were malformed in some way (such as "66##77") so that the corresponding full name if returned would be nonsense then Name_Error is raised. But Name_Error is never raised just because the file does not exist.

On the other hand

```
Simple_Name("c:\adastuff\rat\library.doc")
```

returns "library.doc" and not "rat\library.doc". We can also apply Simple_Name to a string that does not go back to the root. Thus

```
Simple_Name("rat\library.doc");
```

is allowed and also returns "library.doc".

The function Containing_Directory_Name removes the simple name part of the parameter. We can even write

```
Containing_Directory_Name("..\rat\library.doc")
```

and this returns "..\rat"; note that it also removes the separator "\".

The functions Extension and Base_Name return the corresponding parts of a file name thus

```
Base_Name("rat\library.doc")          -- "library"
Extension("rat\library.doc")          -- "doc"
```

Note that they can be applied to a simple name or to a full name or, as here, to something midway between.

The function Compose can be used to put the various bits together, thus

```
Compose("rat", "library", "doc")
```

returns "rat\library.doc". The default parameters enable bits to be omitted. In fact if the third parameter is omitted then the second parameter is treated as a simple name rather than a base name. So we could equally write

```
Compose("rat","library.doc")
```

The next group of functions, Exists, Kind, Size and Modification_Time act on a file name (that is the name of a real external file) and return the obvious result. (The size is measured in stream elements – usually bytes.)

Various types and subprograms are provided to support searching over a directory structure for entities with appropriate properties. This can be done in two ways, either as a loop under the direct control of the programmer (sometimes called an active iterator) or via an access to subprogram parameter (often called a passive iterator). We will look at the active iterator approach first.

The procedures Start_Search, End_Search and Get_Next_Entry and the function More_Entries control the search loop. The general pattern is

```
Start_Search( ... );
while More_Entries( ... ) loop
  Get_Next_Entry( ... );
  ...                          -- do something with the entry found
end loop;
End_Search( ... );
```

Three types are involved. The type Directory_Entry_Type is limited private and acts as a sort of handle to the entries found. Valid values of this type can only be created by a call of Get_Next_Entry whose second parameter is an out parameter of the type Directory_Entry_Type. The type Search_Type is also limited private and contains the state of the search. The type Filter_Type provides a simple means of identifying the kinds of file to be found. It has three components corresponding to the three values of the enumeration type File_Kind and is used by the procedure Start_Search.

Suppose we want to look for all ordinary files with extension "doc" in the directory "c:\adastuff\rat". We could write

```
Rat_Search: Search_Type;
Item: Directory_Entry_Type;
Filter: Filter_Type := (Ordinary_File => True, others => False);
...
Start_Search(Rat_Search, "c:\adastuff\rat", "*.doc", Filter);
while More_Entries(Rat_Search) loop
  Get_Next_Entry(Rat_Search, Item);
  ...                          -- do something with Item
end loop;
End_Search(Rat_Search);
```

The third parameter of Start_Search (which is "*.doc" in the above example) represents a pattern for matching names and thus provides further filtering of the search. The interpretation is implementation defined except that a null string means match everything. However, we would expect that writing "*.doc" would mean search only for files with the extension "doc".

The alternative mechanism using a passive iterator is as follows. We first declare a subprogram such as

```
procedure Do_It(Item: in Directory_Entry_Type) is
begin
  ...                          -- do something with item
end Do_It;
```

and then declare a filter and call the procedure Search thus

```
Filter: Filter_Type := (Ordinary_File => True, others => False);
...
Search("c:\adastuff\rat", "*.doc", Filter, Do_It'Access);
```

The parameters of Search are the same as those of Start_Search except that the first parameter of type Search_Type is omitted and a final parameter which identifies the procedure Do_It is added. The variable Item which we declared in the active iterator is now the parameter Item of the procedure Do_It.

Each approach has its advantages. The passive iterator has the merit that we cannot make mistakes such as forget to call End_Search. But some find the active iterator easier to understand and it can be easier to use for parallel searches.

The final group of functions enables us to do useful things with the results of our search. Thus Simple_Name and Full_Name convert a value of Directory_Entry_Type to the corresponding simple or full file name. Having obtained the file name we can do everything we want but for convenience the functions Kind, Size and Modification_Time are provided which also directly take a parameter of Directory_Entry_Type.

So to complete this example we might print out a table of the files found giving their simple name, size and modification time. Using the active approach the loop might then become

```
while More_Entries(Rat_Search) loop
  Get_Next_Entry(Rat_Search, Item);
  Put(Simple_Name(Item));  Set_Col(15);
  Put(Size(Item/1000));  Put(" KB");  Set_Col(25);
  Put_Line(Image(Modification_Time(Item)));
end loop;
```

This might produce a table such as

```
access.doc        152 KB    2005-04-05  09:03:10
containers.doc    372 KB    2005-06-14  21:39:05
general.doc       181 KB    2005-03-03  08:43:15
intro.doc         173 KB    2004-11-25  15:52:20
library.doc       149 KB    2005-04-08  13:50:05
oop.doc           179 KB    2005-02-25  18:34:55
structure.doc     151 KB    2005-04-05  09:09:25
tasking.doc       174 KB    2005-03-31  11:16:40
```

Note that the function Image is from the package Ada.Calendar.Formatting discussed in the previous section.

Observe that the search is carried out on the directory given and does not look at subdirectories. If we want to do that then we can use the function Kind to identify subdirectories and then search recursively.

It has to be emphasized that the package Ada.Directories is very implementation dependent and indeed might not be supported by some implementations at all. Implementations are advised to provide any additional useful functions concerning retrieving other information about files (such as name of the owner or the original creation date) in a child package Ada.Directories.Information.

Finally, note that misuse of the various operations will raise one of the exceptions Status_Error, Name_Error, Use_Error or Device_Error from the package IO_Exceptions.

5 Characters and strings

An important improvement in Ada 2005 is the ability to deal with 16- and 32-bit characters both in the program text and in the executing program.

The fine detail of the changes to the program text are perhaps for the language lawyer. The purpose is to permit the use of all relevant characters of the entire ISO/IEC 10646:2003 repertoire. The most important effect is that we can write programs using Cyrillic, Greek and other character sets.

A good example is provided by the addition of the constant

```
π: constant := Pi;
```

to the package Ada.Numerics. This enables us to write mathematical programs in a more natural notation thus

```
Circumference: Float := 2.0 * π * Radius;
```

Other examples might be for describing polar coordinates thus

```
R: Float := Sqrt(X*X + Y*Y);
θ: Angle := Arctan(Y, X);
```

and of course in France we can now declare a decent set of ingredients for breakfast

```
type Breakfast_Stuff is (Croissant, Café, Œuf, Beurre);
```

Curiously, although the ligature æ is in Latin-1 and thus available in Ada 95 in identifiers, the ligature œ is not (for reasons we need not go into). However, in Ada 95, œ is a character of the type Wide_Character and so even in Ada 95 one can order breakfast thus

```
Put("Deux œufs easy-over avec jambon");          -- wide string
```

In order to manipulate 32-bit characters, Ada 2005 includes types Wide_Wide_Character and Wide_Wide_String in the package Standard and the appropriate operations to manipulate them in packages such as

```
Ada.Strings.Wide_Wide_Bounded
Ada.Strings.Wide_Wide_Fixed
Ada.Strings.Wide_Wide_Maps
Ada.Strings.Wide_Wide_Maps.Wide_Wide_Constants
Ada.Strings.Wide_Wide_Unbounded
Ada.Wide_Wide_Text_IO
Ada.Wide_Wide_Text_IO.Text_Streams
Ada.Wide_Wide_Text_IO.Complex_IO
Ada.Wide_Wide_Text_IO.Editing
```

There are also new attributes Wide_Wide_Image, Wide_Wide_Value and Wide_Wide_Width and so on.

The addition of wide-wide characters and strings introduces many additional possibilities for conversions. Just adding these directly to the existing package Ada.Characters.Handling could cause ambiguities in existing programs when using literals. So a new package Ada.Characters. Conversions has been added. This contains conversions in all combinations between Character, Wide_Character and Wide_Wide_Character and similarly for strings. The existing functions from Is_Character to To_Wide_String in Ada.Characters.Handling have been banished to Annex J.

The introduction of more complex writing systems makes the definition of the case insensitivity of identifiers, (the equivalence between upper and lower case), much more complicated.

In some systems, such as the ideographic system used by Chinese, Japanese and Korean, there is only one case, so things are easy. But in other systems, like the Latin, Greek and Cyrillic alphabets, upper and lower case characters have to be considered. Their equivalence is usually straightforward but there are some interesting exceptions such as

- Greek has two forms for lower case sigma (the normal form σ and the final form ς which is used at the end of a word). These both convert to the one upper case letter Σ.
- German has the lower case letter ß whose upper case form is made of two letters, namely SS.
- Slovenian has a grapheme LJ which is considered a single letter and has three forms: LJ, Lj and lj.

The Greek situation used to apply in English where the long s was used in the middle of words (where it looked like an f but without a cross stroke) and the familiar short s only at the end. To modern eyes this makes poetic lines such as "Where the bee sucks, there suck I" somewhat dubious. (This is sung by Ariel in Act V Scene I of The Tempest by William Shakespeare.)

The definition chosen for Ada 2005 closely follows those provided by ISO/IEC 10646:2003 and by the Unicode Consortium; this hopefully means that all users should find that the case insensitivity of identifiers works as expected in their own language.

Of interest to all users whatever their language is the addition of a few more subprograms in the string handling packages. As explained in the Introduction, Ada 95 requires rather too many conversions between bounded and unbounded strings and the raw type String and, moreover, multiple searching is inconvenient.

The additional subprograms in the packages are as follows.

In the package Ada.Strings.Fixed (assuming **use** Maps; for brevity)

```
function Index(Source: String; Pattern: String;
          From: Positive; Going: Direction := Forward;
          Mapping: Character_Mapping := Identity) return Natural;

function Index(Source: String; Pattern: String;
          From: Positive; Going: Direction := Forward;
          Mapping: Character_Mapping_Function) return Natural;

function Index(Source: String; Set: Character_Set;
          From: Positive; Test: Membership := Inside;
          Going: Direction := Forward) return Natural;

function Index_Non_Blank(Source: String;
          From: Positive; Going: Direction := Forward) return Natural;
```

The difference between these and the existing functions is that these have an additional parameter From. This makes it much easier to search for all the occurrences of some pattern in a string.

Similar functions are also added to the packages Ada.Strings.Bounded and Ada.Strings.Unbounded.

Thus suppose we want to find all the occurrences of "bar" in the string "barbara barnes" held in the variable BS of type Bounded_String. (I have put my wife into lower case for convenience.) There are 3 of course. The existing function Count can be used to determine this fact quite easily

```
N := Count(BS, "bar")                -- is 3
```

But we really need to know where they are; we want the corresponding index values. The first is easy in Ada 95

```
I := Index(BS, "bar")                -- is 1
```

But to find the next one in Ada 95 we have to do something such as take a slice by removing the first three characters and then search again. This would destroy the original string so we need to make a copy of at least part of it thus

```
Part := Delete(BS, I, I+2);                  -- 2 is length "bar" – 1
I := Index(Part, "bar") + 3;                 -- is 4
```

and so on in the not-so-obvious loop. (There are other ways such as making a complete copy first, this could either be in another bounded string or perhaps it is simplest just to copy it into a normal String first; but whatever we do it is messy.) In Ada 2005, having found the index of the first in I, we can find the second by writing

```
I := Index(BS, "bar", From => I+3);
```

and so on. This is clearly much easier.

The following are also added to Ada.Strings.Bounded

```
procedure Set_Bounded_String(Target: out Bounded_String;
            Source: in String; Drop: in Truncation := Error);

function Bounded_Slice(Source: Bounded_String;
            Low: Positive; High: Natural) return Bounded_String;

procedure Bounded_Slice(Source: in Bounded_String;
            Target: out Bounded_String;
            Low: in Positive; High: in Natural);
```

The procedure Set_Bounded_String is similar to the existing function To_Bounded_String. Thus rather than

```
BS := To_Bounded_String("A Bounded String");
```

we can equally write

```
Set_Bounded_String(BS, "A Bounded String");
```

The slice subprograms avoid conversion to and from the type String. Thus to extract the characters from 3 to 9 we can write

```
BS := Bounded_Slice(BS, 3, 9);                -- "Bounded"
```

whereas in Ada 95 we have to write something like

```
BS := To_Bounded(Slice(BS, 3, 9));
```

Similar subprograms are added to Ada.Strings.Unbounded. These are even more valuable because unbounded strings are typically implemented with controlled types and the use of a procedure such as Set_Unbounded_String is much more efficient than the function To_Unbounded_String because it avoids assignment and thus calls of Adjust.

Input and output of bounded and unbounded strings in Ada 95 can only be done by converting to or from the type String. This is both slow and untidy. This problem is particularly acute with unbounded strings and so Ada 2005 provides the following additional package (we have added a use clause for brevity as usual)

```
with Ada.Strings.Unbounded;  use Ada.Strings.Unbounded;
package Ada.Text_IO.Unbounded_IO is

  procedure Put(File: in File_Type; Item: in Unbounded_String);
  procedure Put(Item: in Unbounded_String);

  procedure Put_Line(File: in File_Type; Item: in Unbounded_String);
  procedure Put_Line(Item: in Unbounded_String);

  function Get_Line(File: File_Type) return Unbounded_String;
  function Get_Line return Unbounded_String;
```

```
  procedure Get_Line(File: in File_Type; Item: out Unbounded_String);
  procedure Get_Line(Item: out Unbounded_String);

end Ada.Text_IO.Unbounded_IO;
```

The behaviour is as expected.

There is a similar package for bounded strings but it is generic. It has to be generic because the package Generic_Bounded_Length within Strings.Bounded is itself generic and has to be instantiated with the maximum string size. So the specification is

```
with Ada.Strings.Bounded;  use Ada.Strings.Bounded;
generic
  with package Bounded is new Generic_Bounded_Length(<>);
  use Bounded;
package Ada.Text_IO.Bounded_IO is

  procedure Put(File: in File_Type; Item: in Bounded_String);
  procedure Put(Item: in Bounded_String);

... -- etc as for Unbounded_IO

end Ada.Text_IO.Bounded_IO;
```

It will be noticed that these packages include functions Get_Line as well as procedures Put_Line and Get_Line corresponding to those in Text_IO. The reason is that procedures Get_Line are not entirely satisfactory.

If we do successive calls of the procedure Text_IO.Get_Line using a string of length 80 on a series of lines of length 80 (we are reading a nice old deck of punched cards), then it does not work as expected. Alternate calls return a line of characters and a null string (the history of this behaviour goes back to early Ada 83 days and is best left dormant).

Ada 2005 accordingly adds corresponding functions Get_Line to the package Ada.Text_IO itself thus

```
function Get_Line(File: File_Type) return String;
function Get_Line return String;
```

Successive calls of a function Get_Line then neatly return the text on the cards one by one without bother.

6 Numerics annex

When Ada 95 was being designed, the Numerics Rapporteur Group pontificated at length over what features should be included in Ada 95 itself, what should be placed in secondary standards, and what should be left to the creativeness of the user community.

A number of secondary standards had been developed for Ada 83. They were

11430 Generic package of elementary functions for Ada,

11729 Generic package of primitive functions for Ada,

13813 Generic package of real and complex type declarations and basic operations for Ada (including vector and matrix types),

13814 Generic package of complex elementary functions for Ada.

The first two, 11430 and 11729, were incorporated into the Ada 95 core language. The elementary functions, 11430, (Sqrt, Sin, Cos etc) became the package Ada.Numerics.Generic_Elementary_

Functions in A.5.1, and the primitive functions, 11729, became the various attributes such as Floor, Ceiling, Exponent and Fraction in A.5.3. The original standards were withdrawn long ago.

The other two standards, although originally developed as Ada 83 standards did not become finally approved until 1998.

In the case of 13814, the functionality was all incorporated into the Numerics annex of Ada 95 as the package Ada.Numerics.Generic_Complex_Elementary_Functions in G.1.2. Accordingly the original standard has now lapsed.

However, the situation regarding 13813 was not so clear. It covered four areas

1 a complex types package including various complex arithmetic operations,

2 a real arrays package covering both vectors and matrices,

3 a complex arrays package covering both vectors and matrices, and

4 a complex input–output package.

The first of these was incorporated into the Numerics annex of Ada 95 as the package Ada.Numerics.Generic_Complex_Types in G.1.1 and the last similarly became the package Ada.Text_IO.Complex_IO in G.1.3. However, the array packages, both real and complex, were not incorporated into Ada 95.

The reason for this omission is explained in Section G.1.1 of the Rationale for Ada 95 [3] which says

> A decision was made to abbreviate the Ada 95 packages by omitting the vector and matrix types and operations. One reason was that such types and operations were largely self-evident, so that little real help would be provided by defining them in the language. Another reason was that a future version of Ada might add enhancements for array manipulation and so it would be inappropriate to lock in such operations permanently.

The sort of enhancements that perhaps were being anticipated were facilities for manipulating arbitrary subpartitions of arrays such as were provided in Algol 68. These rather specialized facilities have not been added to Ada 2005 and indeed it seems most unlikely that they would ever be added. The second justification for omitting the vector and matrix facilities of 13813 thus disappears.

In order to overcome the objection that everything is self-evident we have taken the approach that we should further add some basic facilities that are commonly required, not completely trivial to implement, but on the other hand are mathematically well understood.

So the outcome is that Ada 2005 includes almost everything from 13813 plus subprograms for

- finding the norm of a vector,
- solving sets of linear equations,
- finding the inverse and determinant of a matrix,
- finding the eigenvalues and eigenvectors of a symmetric real or Hermitian matrix.

A small number of operations that were not related to linear algebra were removed (such as raising all elements of a matrix to a given power).

So Ada 2005 includes two new packages which are Ada.Numerics.Generic_Real_Arrays and Ada.Numerics.Generic_Complex_Arrays. It would take too much space to give the specifications of both in full so we give just an abbreviated form of the real package in which the specifications of the usual operators are omitted thus

```
generic
  type Real is digits <>;
package Ada.Numerics.Generic_Real_Arrays is
  pragma Pure(Generic_Real_Arrays);

  -- Types
  type Real_Vector is array (Integer range <>) of Real'Base;
  type Real_Matrix is array (Integer range <>, Integer range <>) of Real'Base;

  -- Real_Vector arithmetic operations
  ... -- unary and binary "+" and "–" giving a vector
  ... -- also inner product and two versions of "abs" – one returns a vector and the
  ... -- other a value of Real'Base

  -- Real_Vector scaling operations
  ... -- operations "*" and "/" to multiply a vector by a scalar and divide a vector by a scalar

  -- Other Real_Vector operations
  function Unit_Vector(Index: Integer; Order: Positive; First: Integer := 1) return Real_Vector;

  -- Real_Matrix arithmetic operations
  ... -- unary "+", "–", "abs", binary "+", "–" giving a matrix
  ... -- "*" on two matrices giving a matrix, on a vector and a matrix giving a vector,
  ... --  outer product of two vectors giving a matrix, and of course
  function Transpose(X: Real_Matrix) return Real_Matrix;

  -- Real_Matrix scaling operations
  ... -- operations "*" and "/" to multiply a matrix by a scalar and divide a matrix by a scalar

  -- Real_Matrix inversion and related operations
  function Solve(A: Real_Matrix; X: Real_Vector) return Real_Vector;
  function Solve(A, X: Real_Matrix) return Real_Matrix;
  function Inverse(A: Real_Matrix) return Real_Matrix;
  function Determinant(A: Real_Matrix) return Real'Base;

  -- Eigenvalues and vectors of a real symmetric matrix
  function Eigenvalues(A: Real_Matrix) return Real_Vector;
  procedure Eigensystem(A: in Real_Matrix;
                        Values: out Real_Vector; Vectors: out Real_Matrix);

  -- Other Real_Matrix operations
  function Unit_Matrix(Order: Positive; First_1, First_2: Integer := 1) return Real_Matrix;

end Ada.Numerics.Generic_Real_Arrays;
```

Many of these operations are quite self-evident. The general idea as far as the usual arithmetic operations are concerned is that we just write an expression in the normal way as illustrated in the Introduction. But the following points should be noted.

There are two operations "**abs**" applying to a Real_Vector thus

```
function "abs"(Right: Real_Vector) return Real_Vector;
function "abs"(Right: Real_Vector) return Real'Base;
```

One returns a vector each of whose elements is the absolute value of the corresponding element of the parameter (rather boring) and the other returns a scalar which is the so-called L2-norm of the vector. This is the square root of the inner product of the vector with itself or $\sqrt{(\Sigma x_i x_i)}$ – or just $\sqrt{(x_i x_i)}$ using the summation convention (which will be familiar to those who dabble in the relative world of tensors). This is provided as a distinct operation in order to avoid any intermediate overflow that might occur if the user were to compute it directly using the inner product "*".

There are two functions Solve for solving one and several sets of linear equations respectively. Thus if we have the single set of *n* equations

$$\mathbf{Ax} = \mathbf{y}$$

then we might write

```
X, Y: Real_Vector(1 .. N);
A: Real_Matrix(1 .. N, 1 .. N);
...
Y := Solve(A, X);
```

and if we have *m* sets of *n* equations we might write

```
XX, YY: Real_Matrix(1 .. N, 1 .. M)
A: Real_Matrix(1 .. N, 1 .. N);
...
YY := Solve(A, XX);
```

The functions Inverse and Determinant are provided for completeness although they should be used with care. Remember that it is foolish to solve a set of equations by writing

```
Y := Inverse(A)*X;
```

because it is both slow and prone to errors. The main problem with Determinant is that it is liable to overflow or underflow even for moderate sized matrices. Thus if the elements are of the order of a thousand and the matrix has order 10, then the magnitude of the determinant will be of the order of 10^{30}. The user may therefore have to scale the data.

Two subprograms are provided for determining the eigenvalues and eigenvectors of a symmetric matrix. These are commonly required in many calculations in domains such as elasticity, moments of inertia, confidence regions and so on. The function Eigenvalues returns the eigenvalues (which will be non-negative) as a vector with them in decreasing order. The procedure Eigensystem computes both eigenvalues and vectors; the parameter Values is the same as that obtained by calling the function Eigenvalues and the parameter Vectors is a matrix whose columns are the corresponding eigenvectors in the same order. The eigenvectors are mutually orthonormal (that is, of unit length and mutually orthogonal) even when there are repeated eigenvalues. These subprograms apply only to symmetric matrices and if the matrix is not symmetric then Argument_Error is raised.

Other errors such as the mismatch of array bounds raise Constraint_Error by analogy with built-in array operations.

The reader will observe that the facilities provided here are rather humble and presented in a simple black-box style. It is important to appreciate that we do not see the Ada predefined numerics library as being in any way in competition with or as a substitute for professional libraries such as the renowned BLAS (Basic Linear Algebra Subprograms, see www.netlib.org/blas). Indeed our overall goal is twofold

- to provide commonly required simple facilities for the user who is not a numerical professional,
- to provide a baseline of types and operations that forms a firm foundation for binding to more general facilities such as the BLAS.

We do not expect users to apply the operations in our Ada packages to the huge matrices that arise in areas such as partial differential equations. Such matrices are often of a special nature such as banded and need the facilities of a comprehensive numerical library. We have instead striven to provide easy to use facilities for the programmer who has a small number of equations to solve such as might arise in navigational applications.

Simplicity is evident in that functions such as Solve do not reveal the almost inevitable underlying LU decomposition or provide parameters controlling for example whether additional iterations should be applied. However, implementations are advised to apply an additional iteration and should document whether they do or not.

Considerations of simplicity also led to the decision not to provide automatic scaling for the determinant or to provide functions for just the largest eigenvalue and so on.

Similarly we only provide for the eigensystems of symmetric real matrices. These are the ones that commonly arise and are well behaved. General nonsymmetric matrices can be troublesome.

Appropriate accuracy requirements are specified for the inner product and L2-norm operations. Accuracy requirements for Solve, Inverse, Determinant, Eigenvalues and Eigenvectors are implementation defined which means that the implementation must document them.

The complex package is very similar and will not be described in detail. However, the generic formal parameters are interesting. They are

```
with Ada.Numerics.Generic_Real_Arrays, Ada.Numerics.Generic_Complex_Types;
generic
  with package Real_Arrays is new Ada.Numerics.Generic_Real_Arrays(<>);
  use Real_Arrays;
  with package Complex_Types is new Ada.Numerics.Generic_Complex_Types(Real);
  use Complex_Types;
package Ada.Numerics.Generic_Complex_Arrays is
  ...
```

Thus we see that it has two formal packages which are the corresponding real array package and the existing Ada 95 complex types and operations package. The formal parameter of the first is <> and that of the second is Real which is exported from the first package and ensures that both are instantiated with the same floating point type.

As well as the obvious array and matrix operations, the complex package also has operations for composing complex arrays from cartesian and polar real arrays, and computing the conjugate array by analogy with scalar operations in the complex types package. There are also mixed real and complex array operations but not mixed imaginary, real and complex array operations. Altogether the complex array package declares some 80 subprograms (there are around 30 in the real array package) and adding imaginary array operations would have made the package unwieldy (and the reference manual too heavy).

By analogy with real symmetric matrices, the complex package has subprograms for determining the eigensystems of Hermitian matrices. A Hermitian matrix is one whose complex conjugate equals its transpose; such matrices have real eigenvalues and are well behaved.

We conclude this discussion of the Numerics annex by mentioning one minute change regarding complex input–output. Ada 2005 includes preinstantiated forms of Ada.Text_IO.Complex_IO such as Ada.Complex_Text_IO (for when the underlying real type is the type Float), Ada.Long_Complex_Text_IO (for type Long_Float) and so on. These are by analogy with Float_Text_IO, Long_Float_Text_IO and their omission from Ada 95 was probably an oversight.

7 Categorization of library units

It will be recalled that library units in Ada 95 are categorized into a hierarchy by a number of pragmas thus

```
pragma Pure( ... );
pragma Shared_Passive( ... );
pragma Remote_Types( ... );
pragma Remote_Call_Interface( ... );
```

Each category imposes restrictions on what the unit can contain. An important rule is that a unit can only depend on units in the same or higher categories (the bodies of the last two are not restricted).

The pragmas Shared_Passive, Remote_Types, and Remote_Call_Interface concern distributed systems and thus are rather specialized. A minor change made in the 2001 Corrigendum was that the pragma Remote_Types was added to the package Ada.Finalization in order to support the interchange of controlled types between partitions in a distributed system.

Note that the pragma Preelaborate does not fit into this hierarchy. In fact there is another hierarchy thus

```
pragma Pure( ... );
pragma Preelaborate( ... );
```

and again we have the same rule that a unit can only depend upon units in the same or higher category. Thus a pure unit can only depend upon other pure units and a preelaborable unit can only depend upon other preelaborable or pure units.

A consequence of this dual hierarchy is that a shared passive unit cannot depend upon a preelaborable unit – the units upon which it depends have to be pure or shared passive and so on for the others. However, there is a separate rule that a unit which is shared passive, remote types or RCI must itself be preelaborable and so has to also have the pragma Preelaborate.

The categorization of individual predefined units is intended to make them as useful as possible. The stricter the category the more useful the unit because it can be used in more circumstances.

The categorization was unnecessarily weak in Ada 95 in some cases and some changes are made in Ada 2005.

The following packages which had no categorization in Ada 95 have pragma Preelaborate in Ada 2005

```
Ada.Asynchronous_Task_Control
Ada.Dynamic_Priorities
Ada.Exceptions
Ada.Synchronous_Task_Control
Ada.Tags
Ada.Task_Identification
```

The following which had pragma Preelaborate in Ada 1995 have been promoted to pragma Pure in Ada 2005

```
Ada.Characters.Handling
Ada.Strings.Maps
Ada.Strings.Maps.Constants
System
System.Storage_Elements
```

These changes mean that certain facilities such as the ability to analyse exceptions are now available to preelaborable units. Note however, that Wide_Maps and Wide_Maps.Wide_Constants stay as preelaborable because they may be implemented using access types.

Just for the record the following packages (and functions, Hash is a function) which are new to Ada 2005 have the pragma Pure

```
Ada.Assertions
Ada.Characters.Conversions
Ada.Containers
Ada.Containers.Generic_Array_Sort
Ada.Containers.Generic_Constrained_Array_Sort
Ada.Dispatching
Ada.Numerics.Generic_Real_Arrays
Ada.Numerics.Generic_Complex_Arrays
Ada.Strings.Hash
```

And the following new packages and functions have the pragma Preelaborate

```
Ada.Containers.Doubly_Linked_Lists
Ada.Containers.Hashed_Maps
Ada.Containers.Hashed_Sets
Ada.Containers.Ordered_Maps
Ada.Containers.Ordered_Sets
Ada.Containers.Vectors
Ada.Environment_Variables
Ada.Strings.Unbounded_Hash
Ada.Strings.Wide_Wide_Maps
Ada.Strings.Wide_Wide_Maps.Wide_Wide_Constants
Ada.Tags.Generic_Dispatching_Constructor
Ada.Task_Termination
```

plus the indefinite containers as well.

A problem with preelaborable units in Ada 95 is that there are restrictions on declaring default initialized objects in a unit with the pragma Preelaborate. For example, we cannot declare objects of a private type at the library level in such a unit. This is foolish for consider

```
package P is
  pragma Preelaborate(P);
  X: Integer := 7;
  B: Boolean := True;
end;
```

Clearly these declarations can be preelaborated and so the package P can have the pragma Preelaborate. However, now consider

```
package Q is
  pragma Preelaborate(Q);                    -- legal
  type T is private;
private
  type T is
    record
      X: Integer := 7;
      B: Boolean := True;
```

```
    end record;
end Q;

with Q;
package P is
  pragma Preelaborate(P);                    -- illegal
  Obj: Q.T;
end P;
```

The package Q is preelaborable because it does not declare any objects. However, the package P is not preelaborable because it declares an object of the private type T – the theory being of course that since the type is private we do not know that its default initial value is static.

This is overcome in Ada 2005 by the introduction of the pragma Preelaborable_Initialization. Its syntax is

```
pragma Preelaborable_Initialization(direct_name);
```

We can now write

```
package Q is
  pragma Preelaborate(Q);
  type T is private;
  pragma Preelaborable_Initialization(T);
private
  type T is
    record
      X: Integer := 7;
      B: Boolean := True;
    end record;
end Q;
```

The pragma promises that the full type will have preelaborable initialization and the declaration of the package P above is now legal.

The following predefined private types which existed in Ada 95 have the pragma Preelaborable_Initialization in Ada 2005

```
Ada.Exceptions.Exception_Id
Ada.Exceptions.Exception_Occurrence
Ada.Finalization.Controlled
Ada.Finalization.Limited_Controlled
Ada.Numerics.Generic_Complex_Types.Imaginary
Ada.Streams.Root_Stream_Type
Ada.Strings.Maps.Character_Mapping
Ada.Strings.Maps.Character_Set
Ada.Strings.Unbounded.Unbounded_String
Ada.Tags.Tag
Ada.Task_Identification.Task_Id
Interfaces.C.Strings.chars_ptr
System.Address
System.Storage_Pool.Root_Storage_Pool
```

Wide and wide-wide versions also have the pragma as appropriate. Note that it was not possible to apply the pragma to Ada.Strings.Bounded.Generic_Bounded_Length.Bounded_String because it would have made it impossible to instantiate Generic_Bounded_Length with a non-static expression for the parameter Max.

The following private types which are new in Ada 2005 also have the pragma Preeleborable_Initialization

Ada.Containers.Vectors.Vector
Ada.Containers.Vectors.Cursor
Ada.Containers.Doubly_Linked_Lists.List
Ada.Containers.Doubly_Linked_Lists.Cursor
Ada.Containers.Hashed_Maps.Map
Ada.Containers.Hashed_Maps.Cursor
Ada.Containers.Ordered_Maps.Map
Ada.Containers.Ordered_Maps.Cursor
Ada.Containers.Hashed_Sets.Set
Ada.Containers.Hashed_Sets.Cursor
Ada.Containers.Ordered_Sets.Set
Ada.Containers.Ordered_Sets.Cursor

and similarly for the indefinite containers.

A related change concerns the definition of pure units. In Ada 2005, pure units can now use access to subprogram and access to object types provided that no storage pool is created.

Finally, we mention a small but important change regarding the partition communication subsystem System.RPC. Implementations conforming to the Distributed Systems annex are not required to support this predefined interface if another interface would be more appropriate – to interact with CORBA for example.

8 Streams

Important improvements to the control of streams were described in the paper on the object oriented model where we discussed the new package Ada.Tags.Generic_Dispatching_Constructor and various changes to the parent package Ada.Tags itself. In this section we mention two other changes.

There is a problem with the existing stream attributes such as (see RM 13.13.2)

procedure S'Write(Stream: **access** Root_Stream_Type'Class; Item: **in** *T*);

where S is a subtype of T. Note that for the parameter Item, its type *T* is in italic and so has to be interpreted according to the kind of type. In the case of integer and enumeration types it means that the parameter Item has type T'Base.

Given a declaration such as

type Index **is range** 1 .. 10;

different implementations might use different representations for Index'Base – some might use 8 bits others might use 32 bits and so on.

Now stream elements themselves are typically 8 bits and so with an 8-bit base, there will be one value of Index per stream element whereas with a 32-bit base each value of Index will take 4 stream elements. Clearly a stream written by the 8-bit implementation cannot be read by the 32-bit one.

This problem is overcome in Ada 2005 by the introduction of a new attribute Stream_Size. The universal integer value S'Stream_Size gives the number of bits used in the stream for values of the subtype S. We are guaranteed that it is a multiple of Stream_Element'Size. So the number of stream elements required will be

S'Stream_Size / Stream_Element'Size

We can set the attribute in the usual way provided that the value given is a static multiple of Stream_Element'Size. So in the case above we can write

```
for Index'Stream_Size use 8;
```

and portability is then assured. That is provided that Stream_Element_Size is 8 anyway and that the implementation accepts the attribute definition clause (which it should).

A minor change is that the parameter Stream of the various atttributes now has a null exclusion so that S'Write is in fact

```
procedure S'Write(Stream: not null access Root_Stream_Type'Class; Item: in T);
```

Perhaps surprisingly this does not introduce any incompatibilities since in Ada 95 passing null raises Constraint_Error anyway and so this change just clarifies the situation.

On this dullish but important topic here endeth the Rationale for Ada 2005 apart from various exciting appendices and an extensive subpaper on containers.

References

[1] ISO/IEC JTC1/SC22/WG9 N412 (2002) *Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of the Amendment*.

[2] ISO/IEC 13813:1997 (1997) Generic packages of real and complex type declarations and basic operations for Ada (including vector and matrix types).

[3] *Ada 95 Rationale* (1995) LNCS 1247, Springer-Verlag.

[4] J. G. P. Barnes (1998) *Programming in Ada 95*, 2nd ed., Addison-Wesley.