Notes on Programs

The programs on this website are generally identical to those in the book apart from layout and the inclusion of considerable additional comments. It is hoped that these additional comments will clarify any details not adequately covered in the text of the book. Note however, that there are two versions of program 6, Playing Pools, as discussed below.

The programs cover a wide range of topics and provide examples of the use of many important Ada constructions. The programs have been limited in size in order to ensure that the book does not become cumbersome. However, the reader might find the following further notes and suggestions for enhancement of interest.

1 Magic Moments

This uses the Ada prefixed notation extensively and some anonymous access types.

This program could be extended in various ways. An obvious extension is to add a further dispatching operation which outputs the values of the additional components of each type. For this to be useful for tabulation it clearly needs to have control of the field width. Thus the width could be a parameter of some Put_Dimensions procedure whose specification might be

```
procedure Put_Dimensions(O: in Object; Width: in Integer) is abstract;
```

In the case of the type Point it might simply output spaces, whereas for the type Triangle it could divide the field into three equal parts and output the size of the three sides accordingly.

Another extension might be to accumulate the sum of the various properties in order to give the total area and the total moments. This could be done within the existing loop in Tabulate_Properties.

For symmetry the moment based on the Y_Coord might also be computed using

```
function Y_Moment(OC: Object'Class) return Float is
begin
  return OC.Y_Coord * OC.Area;
end Y_Moment;
```

and the existing function could then have the identifier X_Moment.

If the totals of these two moments were then computed in Tot_X_Moment and Tot_Y_Moment and the total area in Tot_Area and so on then we could easily compute and print some other properties of the system. (Remember that the x-axis is horizontal and the y-axis is vertical as is usual in calculations involving school mechanics.)

For example, the centre of gravity is at the point given by

```
CG: Point := (X_Coord => Tot_X_Moment/Tot_Area,
              Y_Coord => Tot_Y_Moment/Tot_Area);
```

and so the distance of the centre of gravity from the origin is then Distance(CG).

Those familiar with mechanics might then note that if we now imagine the system as a whole to be pivoted at the origin and allowed to swing then the period of a small oscillation is

```
2 * Pi * Sqrt(Tot_MO / (Distance(CG) * Tot_Area))
```

where Tot_MO is the total moment of inertia about the origin (the sum of the values of MO) - remember that our units are such that *g*, the acceleration due to gravity is 1. Note that Tot_MO / Tot_Area is the square of the radius of gyration. The above formula is simply the familiar

```
T = 2 * pi * sqrt(l/g)
```

for a pendulum where the effective length l is the square of the radius of gyration divided by the distance of the centre of gravity from the origin. Naturally enough, if the centre of gravity is at the origin then the period becomes infinite.

The other kind of extension to consider is simply to add other types representing other kinds of geometrical objects. An obvious possibility is to add a type Regular_Polygon having a discriminant giving the number of sides as in Section 18.4. Good luck with its moment of inertia. Another possibility is to add a type Ellipse derived from Object and then to derive Circle from Ellipse.

As mentioned in the comments, a simple improvement is to make Distance take a class wide parameter; this is discussed further in Chapter 14 - see Exercise 14.3(1).

2 Sylvan Sorter

This uses anonymous access types for the type List but not for the type Tree. It is necessary to apply a type conversion in the procedure Take_From_List and this means that the type List has to be a general access type. No conversions are necessary with the type Tree and so it can be pool specific (and so does not need **all** in its declaration). An interesting challenge is to use anonymous access types for Tree as well as List. This would be tedious in Ada 2005 because of the need for many named conversions but they would not be necessary in Ada 2012.

A possible extension might be to reconsider the algorithm for printing the tree so that better use is made of unused parts of the page.

For example the procedure Do_It might make more intelligent use of the space for a subtree in the case of a parent having only one subtree. This would mean that larger trees could be printed before the numbers degenerated into asterisks.

The simplest approach would simply be to pass on the Col and Width parameters unchanged and this would make a single subtree appear immediately underneath its parent. Clearly the lines would need changing as well and perhaps a symbol could be inserted in order to indicate whether the single subtree is a left or right one.

A more challenging approach might be for subtrees to use the otherwise unused space below siblings of their parent which have no subtrees of their own. This would clearly need a rethink of the whole strategy of the program.

3 Rational Reckoner

This uses a private with clause for Stack.Data on the procedure Stack.Print_Top. There are lots of ways in which this might be extended.

The private child package Rational.Slave could with advantage be replaced by making Normal into a private child function and embedding GCD within Normal (or if the iterative form is used the text of GCD could be just expanded in Normal).

Two extensions mentioned in the book are to issue a prompt and to handle Constraint_Error if the arithmetic overflows.

Using the exception Done as a means of termination might be considered bad practice. An alternative is to pass an indication of the desire to terminate as an **out** parameter of Process. This is straightforward but a bit ugly.

Another obvious extension is simply to add more arithmetic operations like those on a pocket calculator. Thus a number of "memory registers" might be added and so on. Functions such as square root are inappropriate for rational numbers but operations to change sign, duplicate the top item, clear the whole stack, reverse the top two items could be added.

A more difficult extension would be to make the calculation generic with respect to the type of arithmetic being done.

A more exciting extension would be to link the program to some suitable GUI interface so that a picture of the current contents of the stack is continuously displayed. This takes us outside the predefined library of the Ada standard but facilities for doing this are widely available.

4 Super Sieve

This uses anonymous access types for the result of the function Make_Filter and for the variables Next (in Filter) and First (in Do_It).

This program would clearly benefit from a better output interface so that the frames are drawn as described in the book. This just means replacing the body of the child package Eratosthenes.Frame and linking it to a suitable screen interface.

A less ambitious improvement might be to change the existing procedure Make_Frame so that several primes are output on a single line. The obvious way to do this is to give the procedure Put a further parameter denoting the field width so that it becomes

```
with procedure Put(E: in Element; Width: in Integer);
```

and also to pass the value of the required width as a further generic parameter. The procedure Make_Frame can then output as many primes on a line as is appropriate.

Other changes might be to explore other algebras which have primes. Thus the polynomials could have coefficients over other number bases. Indeed the polynomials might even have complex integers as their coefficients!

Other ways of communicating between the tasks and especially of stopping the system might be considered. For example it is possible to arrange the structure so that the terminate alternative makes the subsidiary tasks terminate automatically when the main task has finished. Considerable restructuring is required to do this since at the moment all the tasks are library tasks.

5 Wild Words

A trivial improvement would be for the program to count the number of attempts made before a banana is produced. The average is about 20 with the program as written. Reducing the average word length by making the number of characters in a word equal to 2*Bag.Dice/3 which gives a range of 1 to 8 results in more four lettered words and thus a higher chance of finding Lear and even Puck.

The use of the exception Done to signal that the story has finished is a good example of the use of an exception to unwind several layers even though the end of the story is hardly an exceptional situation. The alternative of passing a parameter from Increment_Line through the intermediate call of Add_Word, Add_Stop or New_Para and then through Process_Paragraph and ultimately to Wild_Words is not attractive.

The text generated by the program is not much like English and there are lots of ways in which it might be improved.

Further tests could be added to Word_OK. For example it could check that no more than three vowels are ever together - indeed few words have more than two adjacent vowels anyway. Perhaps a table of allowed vowel combinations could be added. Similarly, excessive strings of consonants could be forbidden. The letter 'y' poses problems since it can behave as either a consonant or a vowel.

Other tests could ensure that "a" and "an" are used correctly with respect to the next word starting with a vowel. And of course a sentence should never finish with an article or a preposition.

But perhaps some completely different approaches could be tried. One suggested in the book is to work in terms of tokens representing combinations of letters.

Another approach is to use a transition matrix of probabilities which gives the probabilities for one letter to be followed by another letter. This requires a 26 by 26 matrix (or maybe 27 where an extra 'letter' indicates the end of a word). Thus

```
subtype Lets is Character range '`' .. 'z';
TM: array (Lets, Lets) of Float;
```

This uses the grave character which precedes 'a' in the type Character to act as a code for the start and end of a word. So TM('`', 'a') is the probability of the first letter of a word being 'a' and TM('n', 'g') is the probability that an 'n' is followed by a 'g'. Note that since every 'q' is followed by a 'u' then TM('q', 'u') has to have the value 1.0.

Suitable values to initialize the array TM could be obtained by analyzing a large lump of English text.

The random numbers for selecting a letter can now be generated by Float_Random which gives a value in the range 0.0 .. 1.0. So given a letter L, the successor letter is obtained in Next_Letter by a sequence such as

```
Total := 0.0;
R := Random(The_Gen);
for M in Lets loop
   Total := Total + TM(L, M);
   if Total > R then
      Next_Letter := M;
      exit;
   end if;
end loop;
```

where The_Gen is a variable of type Float_Random.Generator. But care is needed that rounding errors in the floating point arithmetic do not cause the loop to finish other than through the exit statement.

6 Playing Pools

This uses the form of raise statement with a message in Allocate and a limited interface for Stack in the root package Stacks.

There are two versions of this program. Program 6 is as in the book but Program 6A is slightly different from that in the book. The main difference is that it provides three types of stack which the user may request by the characters L, D or V.

The third stack type D_Stack uses a linked structure similar to that of the type L_Stack but where the records have a discriminant giving the size of the disc. The records also have an array component whose length is equal to the discriminant and so the records themselves are

of different sizes. The values of the components of the arrays are not actually used but are just gratuitously set to the disc size.

If the type D_Stack is used then the calls of Allocate and Deallocate require different amounts of space according to the disc being moved. With suitable combinations of kinds of stack it is possible to cause fragmentation and so make the program fail and produce the fragmentation message. For example, this should happen with a tower size of 5 and the combination LDL.

Another minor difference is that the exception handler in the main subprogram calls Exception_Message in one and Exception_Information in the other.

A few words of explanation regarding the alignment calculation in Allocate might be helpful.

Note that we make no attempt to align the storage pool itself or the component array Store. It is far simpler just to find out the alignment of the start of the array Store and then compute the required offset to the start of the next correctly aligned lump.

We will now show that the expression

```
Align - Pool.Store(Align)'Address mod Align
```

is indeed the correct value of the index of the first component of the array Store with the required alignment.

If the requested alignment is Align then one of the components from Store(1) to Store(Align) must be correctly aligned because of the cyclic nature of alignment. If it is the component with index Align then clearly the expression above gives the correct value since Store(Align)'Address **mod** Align will be zero.

However, if it is not the correctly aligned one then the required component will be the one with index reduced precisely by the remainder obtained by computing the same expression. Thus in either case the result is correct.

One important improvement that should be made to the program is to check that Alignment is not zero. It shouldn't be zero, but it would be wise to check and raise Error with an appropriate message.

A related point is that perhaps we should raise Storage_Error rather than our own exception anyway.

Major improvements can obviously be made to the searching aspects of the procedure Allocate. The free storage should be linked together in some way so that a free lump can be found quickly; the procedure Initialize can be used for creating the initial linkage. Fancy algorithms such as the well-known buddy algorithm could also be programmed.

Another change might be to add a further discriminant indicating whether monitoring is required or not. This would mean that the considerable space required for monitoring (which is much more than the storage space itself) can be declared in a variant and so only used when necessary. The Boolean array should also be packed.

Finally the reader might consider how to avoid the use of the label Again. It could be argued that the alternatives are not nearly so clear as the brutal use of the poor label. Indeed Ada 2012 is more tolerant of labels as mentioned in Section 7.4.

May 2014