Ada Fundamentals with GNAT: Workshop 3

1. Variant record

Objective: This problem illustrates discriminated record types.

Note: The workshop directory contains the source files for the Person example (slide 6 of Part 3, in Page 3/4 of the course notebook), which will be useful as a model for this exercise.

You have been hired by Dysfunctional Enterprises, Inc., to write an Ada program that models the company’s employment record requirements. Specifically:

a) Record type declaration

Define a package specification Employee\_Pkg containing a record type, Employee, with the following structure:

* Each Employee has a discriminant, Role, of an enumeration type Employee\_Role with the elements Engineer and Manager.
* Each Employee also has an SSN (an Integer), and a Salary (a Positive).
* Each Engineer has a Cubicle\_Size (an Integer)
* Each Manager has an Is\_Pointy\_Haired field (a Boolean)

b) Subprograms

Using the technique shown in the notes (slides 10 and 11 of Part 3, on page 3/6 in the course notebook), define Put and Get procedures for Employee, and define a main procedure that does the following:

* calls Get to prompt for input (the value for each field)
* doubles the Salary and then either increments an Engineer’s Cubicle\_Size by 100 or reverses a Manager’s Is\_Pointy\_Haired status, depending on the Role
* calls Put to display the value for each field

c) Modifications

Add a new variant, Administrative. Each Administrative has a Helpfulness field (an Integer in the range -10 .. 0). Determine the effect of this change on Get, Put, and the main procedure (there is no need to actually make the changes, the point of the exercise is just to give you a feeling of the impact of a maintenance change such as adding a new variant).

2. “Varying-length array”

Objective: This problem illustrates discriminated record types.

A Student record consists of the student’s Name (a Mutant\_String - see slide 14 of Part 3, page 3/8 in the course notebook) and a Grade (an Integer in the range 0..100). A Course consists of a Teacher\_Name (also a Mutant\_String) and an array of Student records. The number of Student records in a given Course can vary, from 0 up to some maximum amount that is established when the Course object is created.

Write a package specification Course\_Pkg with types for Student, array of Student, and Course. (You do not need to provide subprograms for manipulating objects of these types.)

3. Linked Lists

Objective: This problem illustrates “pool specific” access types

The course directory contains a package spec for Int\_List\_Pkg (slide 44 of Part 3, page 3/23 in the course notebook) that includes some additional subprograms for you to implement. The main procedure Int\_List\_Test exercises the new subprograms.

You may assume that the lists are linear (i.e., no cycles), but if you are feeling ambitious you can also make your solutions more robust by having them operate correctly when a cycle is encountered.

a) Traversing a linked list

The Display procedure, when given a linked list, should print the integer data at each node. If the list is **null**, the procedure should simply return.

The Locate function, when given a linked list of integers and an integer N, should return a reference to the first node in the list whose Data field is N, or **null** if there is no such node. Your program should work correctly (returning **null**) when the original list is **null**.

b) Cloning

The Clone function, when given a linked list of integers, should return a new linked list where every node is a copy of the corresponding node in the original list. (Hint: a recursive solution is simpler than an iterative one.)

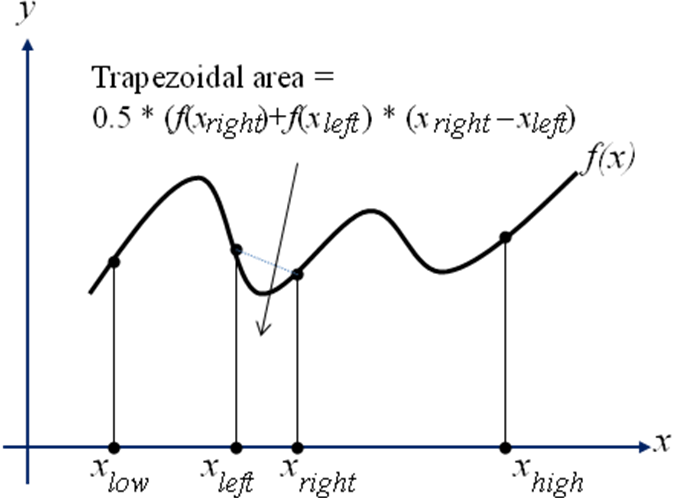
c) Reclaiming complicated data structures (optional)

The package String\_List\_Pkg models linked lists where the Data in each node is an access value designating a String. Implement the subprograms Get, Put, and Reclaim. The Get procedure should prompt the user to enter strings, until an empty string is supplied. Each string is to be inserted into the list that is being built up. (It is easiest to insert the string at the beginning.) Note that what is inserted is actually an access value designating an allocated string initialized to what the user entered. The Put procedure should traverse the list and display the strings that are referenced from the nodes. The Reclaim procedure should reclaim all the allocated objects that are accessible from the list; this comprises the Nodes and also the allocated Strings. When writing Put and Reclaim, you can assume that the list is linear, but as with part (a), feel free to write it more robustly so that it detects cycles.

4. Math integration through a pointer to a function

Objective: This problem illustrates passing a subprogram reference as a parameter

Given a function y = f(x) and endpoints xlow and xhigh, a classical technique for approximating ∫f(x) between xlow and xhigh is by trapezoidal approximation. The approximation function Integrate(f, x\_low, x\_high, n) divides the line between xlow and xhigh into n intervals and adds the areas of the resulting n trapezoids, as illustrated in the figure below.



Declare a package Integrate\_Pkg with an appropriate access-to-function type Func\_Ref (assume that the function takes a Float parameter and returns a Float) and an Integrate function with the specification

**function** Integrate(Func : Func\_Ref;   
 X\_Low, X\_High : Float;  
 N : Positive) **return** Float;

The workshop directory has a main program to exercise this function.

5. “Callback”

Objective: This problem illustrates storing a subprogram reference in a data structure

Generalize Action\_Vector\_Pkg (page 3/40, and in the workshop directory) so that an element of the Action\_Vector array is not just a Func\_Ref but a record containing a Func\_Ref and a Float.

* Implement Action\_Func\_Pkg such that Func1(x) returns x2, Func2(x) returns x/2.0, and Func3(x) returns x + 5.0.
* Modify the main procedure (Action\_Vector\_Test) as follows:
* The Float values in the My\_Action\_Vector entries should be 10.0, 20.0, and 30.0, respectively
* At iteration I the function in Action\_Vector entry I is invoked on the Float item in that entry, and the computed result replaces the Float value.
* In a subsequent iteration, the new Float values in the Action\_Vector entries are displayed

