CS141: Lab 2

1. Overview

How To: In this lab, we'll be writing a program to translate from the *resistor color code* to a simple number. The tools and submission procedures are the same as you used in Lab 1: you'll write the program in AdaGide and copy the source to your submission folder for grading. Section 3.0 gives a general outline of how to start a program from scratch, which should help you with this and all future assignments.

Prerequisites: *Read these notes carefully first.* You should understand how to create programs in AdaGide and submit them for grading. If you successfully completed Lab 1, you should be adequately prepared.

Objectives: To understand how to write programs that perform simple calculations, and that use simple input and output. You should also understand how to work with *enumerated types* (custom *data types* using a list of discrete values, such as colors or days of the week). You should also understand how to plan programs from scratch and write the accompanying pseudocode.

Why This Is Important: All programs involve *input* and/or *output* (collectively called *I/O* or *IO*), and most programs will involve some *operations* on the data in between. Considering a task in terms of *input, output, and operation*s aids in writing and understanding programs.

Many programming languages also allow the programmer to specify their own *data types,* with their own operations (including input and output operations). *Enumerations* are a common custom data type. Enumerations substitute one set of symbols with obvious meanings (such as keywords) for a less-obvious set (such as numbers). You can use enumerations to translate (or *map*) from one set of symbols to another.

Finally, the first homework assignment is very similar to this program. You will find that program a lot easier to write if you thoroughly read, understand, and finish this lab.

Keep In Mind:

- In this program, we'll use a code of ten colors that correspond to the digits 0 through 9. Coincidentally, Ada numbers entries in an enumeration for 0 to 9, so we can easily use Ada *attributes* to get numbers from colors.
- You should get in the habit of identifying the *inputs, outputs, and operations* of both programs and pieces of programs, like procedures and libraries. You should also try to identify the *scratchpads* and *types* you will use early on. Liberally insert *pseudocode* into the program to serve as a scaffolding and guide for your work. It will help you when you need to debug or otherwise make changes.
- Remember what you learned in the first lab. Use the `N:` drive for your *working copy*. Include a *banner* to identify the program and who wrote it. *Rebuild* your program after making changes. *Test* your program to make sure it works. Don't forget to submit your *source code* for grading when you are done.

2. Assignment and Grading

Write a program to translate the *resistor color code* to a number. A *resistor* is a common electronic component. Many resistors are printed with colored bands that encode the number of ohms (a unit of resistance).

The first three colored bands are the digits of the number, and the fourth is the power of ten, so the value is (first-number)(second-number)(third-number) x 10 ^ (fourth-number). The digit for each color is given in the table.

Color	Digit	Color	Digit
Black	0	Green	5
Brown	1	Blue	6
Red	2	Violet	7
Orange	3	Grey	8
Yellow	4	White	9

For example, the color code `Red-Green-Black-Orange` corresponds to 2, 5, 0, and 3, or `250 x 10`^{3}. Also, remember that 10^0 = 1, so (for example) `Brown-Black-Black-Black` corresponds to `100`.

From this description, the I/O and operations should be obvious:

- Inputs: Four colors
- Operations: Map the four colors into the hundreds place, tens place, ones place, and a power of ten.
- Outputs: A single number indicating resistance.

Here is a suggested outline for your program.

1. Define an enumerated type, `ColorCode`, for the colors. As this is a custom type, you will also need to make an I/O handler, based on `Ada.Text_IO.Enumeration_IO`.
2. Ask the user for four colors, read those colors, and store them in scratchpads. You do not have to handle nonexistent colors. For example, if the user inputs `mauve`, it is OK for the program to crash.
3. Find the digits associated with each color. (Possibly) Store these in scratchpads.
4. Print out the resistance, nicely formatted. "Nicely formatted" means that the number reads easily (no extra spaces between the digits). You may use scientific notation. Example: `123 x 10^4`

Test the program. Try several different inputs, including some incorrect ones (eg, enter a color that does not exist.) The program should properly calculate the resistance for any four colors you input. If not, modify the program, re-build, and re-test.

> *Note: When you click Build, wait for the 'Completed Successfully' message to be displayed before trying to run the program, or the program might fail. You must also close the program if it was running (it will be a command line lession, or a"white-on-black typewriter" style window.) If this happens, don't worry; just Build it again and wait a few moments before running.*

When you have tested the program to your own satisfaction, submit the source (eg, the `lab2.adb` file) to your *submit folder*. To keep things from colliding with Lab 1, please create a separate `lab2` sub-folder.

Grading: *A:* Accept the four colors and print the predicted result for all the test values on the last page. *B:* Minor math or formatting errors, but which still read the color correctly. *C:* Doesn't correctly read the colors correctly or calculate the value correctly, but still does meaningful input and output. *D:* Programs that have good source structure and commenting, but which don't compile or run.

3. Procedure

3.0 Data Inside the Computer

If programs are about manipulating data, then it is important to understand exactly how data works inside a computer. The following is a guide to the concepts involved in manipulating data. It will be easier to complete later assignments if you understand these concepts.

Value: A value is a single piece of data in the programming language. It may be a number, a message, or of some user defined type. For example, the value of `2+3` is 5; and the value of the number `5` is also 5. Values can be compared for equality and (usually) to determine which is larger or smaller.

Literal: A literal is a specific value that's hard-coded into the program. Programming languages recognize literal values of their built-in types. For example, 5 is a literal value. `2+3` involves two literal values (a 2 and a 3.) `"Hello World"` is an example of a String literal (a text message). We will deal with Strings later, but you should be aware that every time you `Put` a prompt, it's printing out a literal String.

Variable: A variable is a place where a value is stored. The value stored in a variable may change several times while a program runs, and usually will.

Expression: An expression is a statement that produces a value. There are many different types of expressions. A variable name by itself is an expression (which has the value of that variable). A mathematical formula (such as `2+2` or `MyNumber*10`) is also an expression. It's important to remember that the value of an expression is determined when the statement with the expression in it is executed, not when you write the program (because expressions that contain variables may change values.)

Assignment: An assignment statement copies the value from an expression (on the right) into a variable (on the left.) Assignments are the way that the contents of variables are changes.

Types: Ada is very picky about the types of values. Every value in Ada has a type and can only be used with compatible variables, expressions, etc. Types both restrict the range of data down to a useful subset and provide meaning to raw data (how do you know if `10011101` is a number, a letter, or a collection of bits?)

There are several common built-in types (`Float`, `Integer`, `String`, `Character`). You can define your own, either from scratch, or as subtypes of another type. Strong typing means that you can't directly add a string (text data) to a number, or directly divide a Float by an Integer. Also, you will notice that every Type has its own I/O package (although, thankfully, these packages generally work the same way.)

You may notice that Ada gives compilation errors if you try to assign an Integer literal to a Float (or vice-versa). Unlike some other languages, Ada does not make assumptions about automatic conversion between data types, so it is necessary for you to always specify the fractional part of a Float literal – even if the literal is a whole number. For example, you can't do `3.5+3`, but you can do `3.5+3.0`.

3.1. Defining an Enumerated Type and Creating an I/O Handler

As you might expect, you create an enumerated type by listing all of the possible values. Define an enumerated type, `ColorCode`, for the colors `Black, Brown, Red, Orange, Yellow, Green, Blue, Violet, Grey, White`. These correspond to the digits 0-9.

A new type is created with a `TYPE` statement, which must be placed in the *Declarations* block. The Declarations block is the part of the program that comes between the `Procedure ... Is` and the main body (given by a `BEGIN/END` pair.) The declaration for an enumerated type looks like this:

```
TYPE ColorCode IS (Black, Brown, Red, Orange, Yellow, Green, Blue,
Violet, Grey, White);
```

When you create a new type, you'll usually have to create an I/O handler to go with it. Ada provides a *generic package* that works with any enumerated type. This statement should also go in the declarations block.

```
PACKAGE ColorCode_IO IS NEW Ada.Text_IO.Enumeration_IO(Enum =>
ColorCode);
```

`ColorCode_IO` will now provide `Get` and `Put` procedures so you can do I/O on colors, which means you can type them directly into the keyboard, and print them directly to the screen. (Otherwise, you would have to write your own I/O package just for colors, which is nontrivial and potentially frustrating.)

3.2 Making Scratchpads and Reading the Input Into Them

You need to read four values, which means you will need four scratchpads. The type of the scratchpads should be `ColorCode`. Scratchpads are always of the form *`variable_name : type_name`*. The type may either be a built-in type (like `Integer`) or one you create (like `ColorCode`).

Ada I/O packages use `Put` and `Get` *calls*, which look like the Spider's `TurnRight` and `Step` instructions. A call means that the program jumps to instructions in another place, and then returns as if it had never left, with some useful side-affect. What really happened with the Spider, then, was the program jumped into the Spider package, drew the Spider's movements in the window, and came back.

I/O calls use parameters, which specify what *values* the call will operate on. We'll talk more about values in the next lab; for now, just think of them as scratchpads for calls. Most often, parameters have names. The most common named parameter to an I/O call is named `Item`. There are a lot of variations on the basic theme; see your textbook or the Ada language reference for details.

Examples of scratchpads, `Put`, and `Get` are shown here. ***Important:*** Do not confuse variable names with type names. it doesn't make sense to read a color to `ColorCode`; rather, it makes sense to read a color to something with has the type `ColorCode`. Make a scratchpad of your type and operate on it, instead.

```
-- in Declarations
  Band1 : ColorCode;
-- Later, in the body
  Ada.Text_IO.Put("Please Enter a Color > ");
  ColorCode_IO.Get(Item => Band1); -- Reads a color
  Ada.Text_IO.Put("You entered: ");
  ColorCode_IO.Put(Item => Band1); -- Prints the same color
  ColorCode_IO.Get(Item => ColorCode); -- Wrong
```

In our programs, we always Get from the keyboard or a file, and Put to the screen or a file. The screen is limited to a simple format like lines on a printed page. (We'll cover files later; they're much the same, except that the page can be as long as necessary.)

As you might guess, since the input is line-oriented, you can enter a value by typing it in and pressing the

Enter key. The preceding code fragment will result in a conversation something like this:

```
Please enter a color > red
You entered: RED
```

For more specifics on the behavior of Ada's text IO system, refer to the text book; among other things, the value was capitalized, in keeping with Ada's generally case-insensitive nature. These notes will elaborate on the text IO system only when the behavior is non-obvious.

3.3 Making the Calculation (Or, Doing Math Using your Enumerated Type)

As mentioned, to turn the color code into a number, simply use the first three colors as the digits, and multiply by a power of ten, given in the fourth digit. To put it another way, the resistance is ((*First* * 100 + *Second* * 10 + *Third*) * 10^{Fourth}).[1]

To get digits from the scratchpads, use the Ada position attribute, `'Pos`. As it happens, Ada begins numbering elements with 0, just like our color code. Thus, as long as you gave the color values in the same order as the table, it's easy to get the corresponding numerical value. The `'Pos` operator is used on the type, and given the value or scratchpad as an (un-named) parameter. For example:

```
-- In declarations
Digit1, Digit2: Integer;
Band1: ColorCode;
-- Later in the body
Band1 := Blue;
Digit1 := ColorCode'Pos(Band1); -- 6
Digit2 := ColorCode'Pos(Blue); -- Also 6
```

The code fragment demonstrates the assignment operator, :=, which copies the value from the right side, to the scratchpad on the left. The value on the right may be a literal (such as `Blue` or `12345`), or it may be a calculation (such as `2 + 2` or `Digit1 * 10`), or it may be another scratchpad; the next lab will talk more about variables, values, scratchpads, and types.

This code fragment also demonstrates working with `Integer` as a type; integers are simply whole numbers (numbers with no fractional parts.) There is a corresponding type, `Float`, for numbers that have fractional parts, and several types of text data which we'll cover in later labs.

You don't always want to use Floats for numbers, even though they may sound like they are more capable. The reason is that Integers are usually faster in the computer, and don't suffer from rounding errors the way that Floats do. On the other hand, on many computers, Integers are limited to numbers between 2^{-31} and 2^{+31}(about two billion), so they won't necessarily hold the entire range of the color code, either!

3.4 Printing the Output

Now that you have the digits, you need to make a *design decision* about how to print the output. A design decision occurs when there is more than one way to proceed, and it is up to the programmer to decide.[2]

There are four obvious methods for printing the output; any are acceptable for this lab.

1. Ada's scientific notation. Calculate the resistance value as a float, and print using the package `Ada.Float_Text_IO` (check the textbook for details about this package.)
2. Integer notation. Calculate the resistance value as an integer and print using the package `Ada.Integer_Text_IO`. If you choose this method, you won't be penalized for failing to handle resistances about 2 billion; but you should explain in pseudocode why this limitation occurs.

1 Remember that, in Ada arithmetic, `*` means multiplication, and that `**` means exponent (“To the power of”.)

2 Personal preference, or at least, personal familiarity, is often a factor in design decisions; in other words, it's okay to “go with what you know,” even if it's not the fastest, cheapest, or prettiest option.

3. Do-it-yourself scientific notation. Print three digits, a text message indication “x 10 ^”, and then print the fourth digit.
4. Integer math and do-it-yourself scientific notation. Calculate the first three digits of the resistance and print it as in (2), and then print the fourth digit as the exponent as in (3).

Depending on the method you have chosen for calculating the resistance, you will use some combination of `Ada.Text_IO`, `Ada.Integer_Text_IO`, and/or `Ada.Float_Text_IO` to print the number. As with the Spider package, these packages must be imported at the top of the program using a `With` statement. The following is a code fragment for printing with Integer_Text_IO. This fragment also demonstrates that you can print a literal value, or a scratchpad.

```
WITH Ada.Integer_Text_IO;
-- Later, in declarations
i1 : Integer;
-- Later, in the body
-- Prints '   12345'
Ada.Integer_Text_IO.Put(Item => 12345);
-- Prints '12345'
i1 := 12345;
Ada.Integer_Text_IO.Put(Item => i1, Width => 1);
```

`Ada.Integer_Text_IO.Put` offers an additional parameter, Width, which specifies the minimum size to use when printing an Integer. The default is 8. If you are getting extra spaces in your output, try using `Width => 1`, as above.

3.5 Testing the Program

Test the program under a range of inputs. This means running the program many times, giving the inputs, and checking that the output matches what you expect. You can (and should) also try 'bad' inputs, to see what will happen if you give a non-color. This table gives some example inputs and the expected outputs; feel free to try others:

First	Second	Third	Fourth	Result
Red	Green	Black	Blue	250,000,000
Blue	Red	Red	Black	622
Green	Black	Black	Red	50,000
Apple	Grape	Banana	Cherry	Failure (Apple isn't a color)
Green	Black	Black	Violet	5,000,000,000 (May not work – see 3.3)