Baby steps in a...

BIG WORLD

Ada, a language for everyone

By Luke A. Guest

Introduction

In this article I will introduce the Ada programming language, its history, what it can be used for and also, how you can use it with your Raspberry Pi computer. This article has a number of coloured side-boxes which provide extra information as I introduce Ada, you should read these so you gain a deeper understanding of the language. So let's get started...

You probably know what other programming languages look like, most of these are not very readable and seem to be a jumble of words (sometimes, not even words) and weird symbols. All languages provide various symbols, but Ada was designed so that its programs were easier to read by other people, even many years after they were originally written.

Unlike Python or Ruby, Ada is a compiled language, much like C. This means we have to pass Ada programs (source) through something called a compiler which converts this source into machine language so that it can be run directly on the computer.

In this article, I will be using a Debian based system image, so Debian Squeeze or Raspbian will be fine. Debian provides an Ada compiler, if you are using a different Linux distribution such as Fedora, you will have to check their package managers for the compiler. The compiler is called GNAT so you know what to look for.

Before we get started, I will assume you are running a graphical environment, such as LXDE (after typing startx or booting directly into it, see MagPI issue 3 page 3 for more info), even though in this article we will be using the console only to start.

I have tested the examples in this text by logging into a remote shell on my Raspberry Pi.

Getting started

Before we can start typing in Ada code and running it on the computer, we need to install a few tools. We will need the terminal, a compiler and an editor, start LXTerminal and type in the following commands to install the tools we need:

```
$ sudo apt-get install gnat
```

You will be asked for your password, enter this, when APT asks you if you want to continue press the return key at this point to let APT install its packages.

Then create a directory for this article's source, we will need to change to this directory as we will be running commands directly inside it:

```
$ mkdir -p \
  $HOME/src/baby_steps/lesson1
$ cd $HOME/src/baby_steps/lesson1
```

Let us create a new Ada source file in this window by typing the following in the shell:

```
$ nano -w hello.adb
```

Inside nano, type in the program in Listing 1 (without the line numbers), typing Ctrl+O to save the program.

Now inside LXTerminal, create a new terminal tab with Ctrl+Shift+T, this will automatically make the new shell's current directory be the same one we are using for this program. We can now compile this program with the following shell command:

```
1   with Ada.Text_IO;
2   use Ada.Text_IO;
3
4   --  Print a message out to the screen.
5   procedure Hello is
6   begin
7      Put_Line ("Hello, from Ada.");
8   end Hello;
```

Listing 1: hello.adb

Line 4: starts with 2 hyphens (or dashes), this is a comment. Anything placed after the hyphens is ignored by the compiler up to the end of the line.

In Ada, a main program can be called almost anything you like, but the filename must match this name (in lower case letters) and have ".adb" appended to the name. In our example, our main program is called "Hello" (lines 5 and 8) and its filename is "hello.adb," the adb means "Ada Body."

Line 5 states our program is a procedure, this is 1 type of Ada's subprograms, the other being a function. Both types of subprogram are used for specific reasons, a procedure does not return any values to the caller whereas a function does. "main" subprograms are procedures.

Line 7 is a call to a subprogram found within a package called Ada.Text_IO, lines 1 and 2. The Put_Line procedure prints whatever is in the string (between double quotes ") to the console.

Line 1 states that we wish to use the facilities provided by the Ada.Text_IO package, and Line 2 tells the compiler we don't want to have to write the subprogram call in full, in other words, if line 2 didn't exist we would have had to type in Ada.Text_IO.Put_Line. There are reasons to do this, but we will cover this another time.

As all subprograms must have a beginning (line 6), they must also have an ending, line 8. In Ada, all subprograms must state what it is ending by specifying its name again. Ada enforces this as this is an aid to being more readable.

Each Ada program is made up of a number of statements, a statement ends with a semi-colon (;). Every statement you write must have a semi-colon otherwise the program will not compile.

In Ada, there are some words which are defined by the language, these are called keywords, you cannot use these keyword names for your own types, variables or subprograms.

```
$ gnatmake hello
```

The program, gnatmake, is the front end to the Ada compiler, as you will see when you compile the program, it calls other programs, including gcc, gnatbind and gnatlink.

You can now run the compiled program with the following command:

```
$ ./hello
```

The result is that the program prints what is in the double quotes in the program to the shell, in other words, "Hello, from Ada." You have just written your first Ada program!

Simple types and maths

Unlike other languages, such as C, Ada is a what is called a strongly typed language. What is this type thing? Well, every value in Ada has a type, for example, the number 10 is an integer number, so if we wanted to store values of numbers we would define a variable of type integer, see the sidebar for more information on types. Exit nano using CTRL+X and create a new file called simple_types.adb and type in the source from Listing 2.

Using what you learnt from the previous example, type in and save this code, then compile it with gnatmake and finally, run the

```
1    with Ada.Text_IO;
2    use Ada.Text_IO;
3
4    procedure Simple_Types is
5       X      :          Integer := 10;
6       Y      : constant Integer := 20;
7       Result :          Integer := 0;
8    begin
9       Result := X + Y;
10
11      Put_Line ("Result = " & Integer'Image (Result));
12   end Simple_Types;
```

Listing 2: simple_types.adb

So, we've already seen lines 1, 2, 4, 8 and 12. So what's new? We have not seen the variable and constant definitions before, these are on lines 5, 6 and 7. Here we define 2 variables, X and Result which are integer types and 1 constant, Y, which is also an integer type.

The difference between a variable and a constant is that you can assign a value to a variable within the program, see line 9, where we assign X + Y to Result. In Ada the symbol := means assign or give the variable on the left the value of what is on the right, in our case this is 10 + 20 which makes Result equal 30. To make something constant we use the keyword "constant" before the type name (integer).

So what happens if you try to assign a value to Y in the program source? Try this yourself and see what happens when you compile the program. It will not compile, because you cannot assign to a constant once it has already been assigned to.

On line 11, there is something strange Integer'Image. What is this? This is an attribute of Integer. See the sidebar entitled "Cool features: Attributes" for more on these.

Also, on line 11, we have another symbol, &, which means string concatenation. This means we can "add" strings together, the left side of & is added to the right side of & and then Put_Line prints everything to the screen.

program in the terminal and see what happens.

Exercises

1. Change line 9 to each of the following, compile and run, what is the value of Result?

   a) X – Y
   b) Y - X
   c) X * Y
   d) X / Y
   e) Y / X

2. Use this time to play around with various numbers, variables and constants and see what results you get in the console.

Cool features: Types

Types enable the compiler to make sure that only variables of the same type can be used together, for example, X := Y + 10; X, Y and 10 are all integers, if X was something else, say of type Boolean, this program would not make sense and it would not compile; the compiler would give you a very helpful message to find your error.

Numeric types

Along with Integer types there are two more types which are based on the Integer type called Natural and Positive types; these are subtype's of Integer in that they restrict the range of values allowed to be assigned to variables of these types.