Ada
A quick crash course

Patrik Broman

September 8, 2014

Abstract

This is a quick course for someone who knows how to program, and needs to quickly understand the basic syntax of Ada.

Contents

1 Goals of this tutorial

The purpose is to prepare a student for a course that requires some programming experience and uses Ada as the programming language. After this course the student should be able to understand the basic syntax of the language and, utilizing previous knowledge, be able to create more advanced programs in Ada.

2 Prerequsites

The student is assumed to have some experience with an imperative programming language, such as Python, C/C++, Java, Pascal etc. More specifically, the student is assumed to be familiar with concepts like variables, functions, branching, loops and other basic concepts of programming.

3 Other assumptions

It's assumed that a vast majority of the ones reading this tutorial already knows C/C++ or Java. With respect to that, some examples will be shown in both C/C++ and Ada to show the differences. Java and C/C++ are very similar in syntax, so if you're a Java programmer it should not matter. Please note that knowledge in one of these languages is not required to understand this course, but for most of the examples comparations with C/C++ is made.

4 What is Ada?

Ada is an imperative programming language. It supports OOP and is statically typed. The syntax is very similar to Pascal. What really is significant for Ada is the built-in support for multithreading, synchronisation, resource protection and such things. The language is not case sensitive.

5 Ada

5.1 Hello World!

Hello World!

Yes, we follow traditions. Here is the classic program in Ada:

```
-- hello.adb
with Ada.Text_IO;
```

```
use Ada.Text_IO;

procedure Hello is
begin
  Put_Line ("Hello world!");
end Hello;
```

Let's have a look at each row. The first row is simply a comment. Comments starts with two dashes. The second row corresponds to *#include < stdio.h >* in C and *#include < iostream >* in C++. The next row corresponds to namespace std in C++.

Then we declare a procedure, wich is the same as a fuction with void as return type. Code blocks are created with the keywords begin and end instead of curly braces. You may notice that the name of the procedure appears at the last line. This is not required for procedures and functions, but recommended. When declaring tasks it is required though. It is also required for ending if statements and such. Here is the same program again, but with corresponding C++ code in comments:

```
-- hello.adb                        --  // hello.cpp
with Ada.Text_IO;                   -- #include <iostream>
use Ada.Text_IO;                    -- using namespace std;

procedure Hello is                  -- void main()
begin                               -- {
  Put_Line ("Hello world!");        --   cout << "Hello world!" << endl;
end Hello;                          -- }
```

5.2 Basic structure of a program

An Ada source file contains one compilation unit. Declaring different procedures in one file will not work. This is not a valid program:

```
-- notvalid.adb
with Ada.Text_Io;
use Ada.Text_Io;

procedure Foo is
begin
   Put_Line("Foo");
end Foo;

procedure notvalid is
```

```
begin
    Put_Line("Not valid");
end notvalid;
```

The compiler outputs this error message:

gcc-4.9 -c notvalid.adb notvalid.adb:9:01: end of file expected, file can have only one compilation unit gnatmake: "notvalid.adb" compilation error

There are different ways to get around this. One way is to put the procedure Foo in a separate file called foo.adb and add the statement with foo; to notvalid.adb. Another way is to declare foo inside notvalid, as shown below:

```
-- valid.adb
with Ada.Text_Io;
use Ada.Text_Io;

procedure valid is
    procedure Foo is
    begin
        Put_Line("Foo");
    end Foo;

begin
    Put_Line("Valid");
end valid;
```

Naming the file to the compilation unit is not required but the compiler will generate a warning unless you do. And yes, we changed the name to valid for obvious reasons.

5.3 Variables

In Ada all variables a procedure or function will be using can be declared before the begin statement. Here is a program that asks for two numbers and prints the sum:

```
-- sum.adb
with Ada.Text_Io; use Ada.Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;

procedure Sum is
    A,B,C : Integer;
begin
    Put("First number: ");
    Get(A);
    Put("Second Number: ");
```

```
    Get(B);
    C:=A+B;
    Put("The sum is ");
    Put(C);
end Sum;
```

As you can see the variables are declared before the actual code. It's also possible to declare the variables inside the code block, using the keyword declare. There are many different types. In general, their names is the same as in C++, but withoug acronyms. Instead of int, you type ingeger. Instead of char, you type character and so on. Declaring a variable follows this syntax:

```
<name>[, <name>...] : <type> [:= <value>];
```

In it's most basic form, you just declare one variable and chose the type. Here is an example:

```
foo : Integer;
```

You may also declare several variables at once:

```
foo, bar, foobar : Integer;
```

Another thing you can do is to initialize a variable upon declaration:

```
foo : Integer := 1337;
```

If you want the variable to be constant, you add the constant keyword:

```
foo : Constant Integer := 1337;
```

5.4 Operators

The operators works about the same as in C++, but some are slightly different. Assignment is performed with := instead of =. To compare values = is used instead of ==.

5.5 Functions and procedures

Functions and procedures are the same thing. The only difference is that functions returns something, while a procedure don't. A procedure looks like this:

```
procedure <name> [(<arguments)] is
<variable declarations>
begin
<code>
end name;
```

If the procedure (or functions) does not take any arguments the parentheses are omitted both in declaration and upon calling. A procedure that takes three integers as argument and stores the sum of the first two variables in the third looks like this¿

```
procedure Sum(A,B : in Integer; C: out Integer) is
begin
  C:=A+B;
end Sum;
```

If you want a function that takes two integers and returns the biggest it would instead look like this:

```
function Max(A,B : in Integer) return Integer is
begin
  if A>B then
    return A;
  else
    return B;
end Max;
```

The keywords in and out determines if the arguments are used to be read or written to. If none is given, in will be assumed. You can't write to a variable unless out is specified. You can read from an out variable, but unless you have initialized it inside the function it will only contain garbage, even if the variable is initialized outside. So if you want to write a function that increments the value by one both in and out are needed:

```
procedure Inc(A: in out Integer) is
begin
  A:=A+1;
end Inc;
```

It should be mentioned that the keyword in is also used for membership of sets.

5.6 Loops

5.6.1 Endless loop

```
loop
-- code
end loop;
```

5.6.2 While loop

```
while <condition> loop
-- code
end loop;
```

5.6.3 For loop

```
for i in Integer range 1..10 loop
--code
end loop;
```

We stop here and take a look at the first row. A for loop in Ada works on lists. We here create the loop variable i and let it have all the values of integer 1 to 10. If you are used to Python or Matlab this should not be so strange, but it's quite different from C. The above code can be shortened to:

```
for i in range 1..10 loop
-- code
end loop;
```

Arrays can be easily looped. If X is an array, you can loop over all elements like this:

```
for i in X'Range loop
--code
end loop;
```

It should be mentioned that loop variables can't be changed inside the loop. This code is not valid:

```
for i in 1..10 loop
  i:=i+1;
end loop;
```

5.6.4 Do-while loop

```
loop
--code
exit when <condition>
end loop;
```

but you can also have code after the exit statement:

```
loop
--code
exit when <condition>
--code
end loop;
```

In that sense, this loop is not really like a do-while. It's more like a $while(true)\{if(<condition>)break;\}$

Finally, you can name loops, but if you do, you have to use the name after the end loop statement, and you may use it after exit.

```
my_loop:
loop:
--code
exit my_loop when <condition>  -- my_loop may be omitted
--code
end loop my_loop;  -- my_loop may not be omitted
```

6 Legal