Ada

LP - Trabajo Dirigido

Autor: 4613A2

Índice

1 Introducción

Ada es un lenguaje de programación multiparadigma con el propósito de hacer los programas de este lenguaje con un gran nivel de seguridad y fiabilidad. Utilizado en entorno tan críticos como defensa, aeronáutica (Airbus o Boeing utilizan *Ada*), gestión de trafico aéreo, entre otros.

2 Historia

Durante los inicios de los años 70. El departamento de defensa de los estados Unidos (*DoD*) buscaba una solución a la gran proliferación de lenguajes de programación que se estuvieron haciendo durante los años 60 y principios de los 70, dando como problema la poca portabilidad de los proyectos y la poca reutilización de estos [9]. Por lo tanto, durante Agosto de 1977 y Marzo de 1979, la *DoD* establece una competición para poder crear este lenguaje. Al principio de este, se establecieron 4 equipos:

- *CII Honeywell Bull* (Verde)
- *Intermetrics* (Rojo)
- *SofTech* (Azul)
- *SRI* (Amarillo)

Estos estaban identificados por un color para no tener ningún tipo de preferencia a la hora de elegir al ganador. Al final, el equipo ganador fue el Verde (*CII Honeywell Bull*) y se le encargo a un equipo liderado por el *Dr. Jean Ichbiah*. Una cosa a destacar de este proceso de selección es que no solo intervino la *DoD* sino también la *OTAN* en todo el proceso de selección.

El nombre *Ada* viene de la condesa de *Lovelace*, *Augusta Ada Byron*, la cual es considerada la primera programadora del mundo, gracias a la relación con *Charles Baddage* creador de la *Maquina Analítica* [2].

Figure 1: Retrato de Ada Lovelace

3 Paradigmas de programación

Ada es un lenguaje multiparadigma, a continuación mostraremos los paradigmas más importantes que maneja [1].

3.1 Programación Orientada a Objetos

Aunque el propósito de *Ada* no se enfoco en este paradigma si que llega a soportar las características de este. A diferencia de otros lenguajes de programación, como *C++* o *Java*, donde se hace el uso de la palabra `class`, *Ada* ofrece un sistema basado en tipos y paquetes donde se nos ofrece un entorno similar a una clase. De una manera análoga a otro lenguaje de programación. Las clases de Ada se pueden ver como `structs` y funciones que manipulan estas `structs` dentro de un archivo externo (archivo *.cpp*) con la peculiaridad de que los atributos de la `struct` son privados.

A pesar de esto, *Ada* intenta hacer que su sistema de clases se parezca a lo que el programador promedio está acostumbrado a hacer. Como la introducción de la notación *"."* en la llamada de métodos entre otras características existentes [7].

A continuación veremos un ejemplo de clase en *Ada*:

```
    -- Especificación de la Clase Pila de Enteros
generic
   Max_Size : Positive;
package Pila is
   type Stack is private;
   procedure Push(Item : in Integer; S : in out Stack);
   -- Agrega un elemento a la pila.
   procedure Pop(S : in out Stack);
   -- Elimina el elemento superior de la pila.
private
   type Stack is tagged record
      Data : array (1..Max_Size) of Integer;
      Top_Index : Natural := 0;
   end record;
end Pila;
```

Figure 2: Ejemplo de la especificación de una clase *Pila de enteros*

```
package body Pila is
   procedure Push(Item : in Integer; S : in out Stack) is
   begin
      if not Is_Full(S) then
         S.Top_Index := S.Top_Index + 1;
         S.Data(S.Top_Index) := Item;
      else
         -- Pila llena.
         null;
      end if;
   end Push;

   procedure Pop(S : in out Stack) is
   begin
      if not Is_Empty(S) then
         S.Top_Index := S.Top_Index - 1;
      else
         -- Pila vacía.
         null;
      end if;
   end Pop;
end Pila;
```

Figure 3: Implementación de la clase *Pila de enteros*

```
with Ada.Text_IO; use Ada.Text_IO;
with Pila;
procedure Main is
   package Int_Stack is new Pila (Max_Size => 10);
   My_Stack : Int_Stack.Stack;
begin
   -- Pushing an element onto the stack
   Int_Stack.Push(42, My_Stack);
end Main;
```

Figure 4: Ejemplo de uso de la clase *Pila de enteros*

A continuación, podemos ver las principales ventajas que nos ofrece la programación orientada a objetos:

- **Reutilización de código**: Gracias al concepto de herencia permite que clases puedan heredar atributos y métodos. Proporcionando así un mecanismo tanto de reutilización de código, como de extensión de este.
- **Encapsulamiento**: El encapsulamiento en este caso lo hacemos gracias al sistema de paquetes que tiene *Ada*. Gracias a esto logramos ocultar detalles internos de la clase y dejando ver solo una interfaz pública. Esto ayuda a poder hacer código de manera más segura y ayuda al mantenimiento del código.
- **Organización de código**: Lo que supone una ayuda en la legibilidad y claridad de este.
- **Herencia**: La herencia promueve la fácil reutilización del código a su ve que permite una organización jerárquica de este.

Como podemos ver, el paradigma de programación orientada a objetos nos ofrece un gran ayuda sobretodo en la gestión de código y a su vez genera una metodología de creación de este. Todo esto ayuda a la disminución y detección de errores. Ideas que están muy relacionadas con la filosofía del lenguaje.
Aún así, este paradigma no ha estado siempre en el lenguaje. No fue hasta Ada 1995 con el auge de los lenguajes de programación orientados a objetos, que no se llego a hacer esta implementación que llega a tener este sistema de manipulación de tipos.

3.2 Procedural

Ada tiene la capacidad de crear procedimientos y funciones que ayudan a organizar el código de una manera estructurada. Una de las características que tiene este lenguaje, es la capacidad de crear de crear *procedimientos anidados*. Un ejemplo de declaración de procedimientos puede ser la figura 4.

3.3 Programación paralela y concurrente

Ada tiene soporte a la programación concurrente desde su primera versión. Para llevar a cabo este paradigma, hace uso del concepto de *tarea*, muy similar a `function` o `procedure` [5]. A continuación dejamos un ejemplo de código:

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is
   task My_Task;
   task body My_Task is
   begin
      for I in Character'('A') .. Character'('Z') loop
         Put_Line("TAREA");
         Put_Line (Character'Image(I));
      end loop;
   end My_Task;
begin
   for I in Character'('A') .. Character'('Z') loop
      Put_Line (Character'Image(I));
   end loop;
end Main;
```

Figure 5: Creación de una tarea donde se hace concurrentemente tanto *My_Task* como el procedimiento *Main* [5].

A su vez, vemos que una de las principales características de *Ada* es el énfasis que hay en todo el mecanismo tanto de sincronización y comunicación entre procesos. A este mecanismo se le llama *Rendezvous*. A continuación se muestra un ejemplo sencillo donde se utiliza lo mencionado previamente:

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is
   task After is
      entry Go;
   end After ;
   task body After is
   begin
      accept Go;
      Put_Line ("After");
   end After;
begin
   Put_Line ("Before");
   After.Go;
end;
```

Figure 6: Ejemplo de utilización de un mecanismo de sincronización [5].

En este caso, gracias a especificar `entry Go` en la tarea nos permite crear un punto de sincronización desde fuera de esta. El código de la tarea estará esperando en la instrucción `accept Go;` hasta que otro proceso externo haga uso de Go. Haciendo que primero salga por consola la palabra *Before* y posteriormente *After*. Por último, en el caso de que *Main* haya acabado y la tarea *After* no haya acabado, *Main* se suspenderá hasta que la tarea haya finalizado [5].

Por lo tanto, vemos que *Ada* tiene un sistema de gestión de los diferentes procesos bastante robustos donde permite que las tareas pueden ser controladas en todo momento por el procedimiento que los crea y no permite que procedimientos padres acaben antes que sus tareas creadas.

3.4 Programación Genérica

A diferencia de lenguajes como *C++* o *Java* donde se utilizan *templates* a nivel de clase. *Ada* lo implementa a nivel de paquete. A su vez, también lo implementa a nivel de rutina/subprograma [6].

La principal diferencia respecto a otros lenguajes es que se tiene que instanciar explícitamente todo lo que tenga el uso de genéricos.

4 Relación con lenguajes de programación similares

Debido a que *Ada* fue creado por el departamento de defensa de los estados unidos. Este se basó en experiencias anteriores con lenguajes de programación y se diseñó para abordar

las limitaciones y deficiencias que la gente percibía de estos lenguajes. A continuación se muestran lenguajes que llegaron a influenciar a *Ada* [2]:

4.1 Pascal

Pascal ha sido una de los lenguajes de programación que más a ha influido en lenguajes posteriores que si que alcanzaron una gran difusión. Como *Java*, *C*, entre otros.
En el caso de *Ada*, la principal característica el cual es influencia de *PASCAL* es su sintaxis estructurada (basada en bloques y declaraciones). A continuación podemos ver una comparación de un mismo código, tanto en *Ada* como *Pascal*.

```
program Factorial;
function Calcular_Factorial(N: Integer): Integer;
begin
   if N <= 0 then
      Calcular_Factorial := 1
   else
      Calcular_Factorial := N * Calcular_Factorial(N - 1);
end;
var
   Numero: Integer;
begin
   Write('Ingrese un número para calcular el factorial: ');
   Readln(Numero);
   Writeln('Factorial de ', Numero, ': ', Calcular_Factorial(Numero));
end.
```

Figure 7: Ejemplo de factorial calculado en *Pascal*

```ada
with Ada.Text_IO; use Ada.Text_IO;
procedure Factorial is
   function Calcular_Factorial(N : Integer) return Integer is
   begin
      if N <= 0 then
         return 1;
      else
         return N * Calcular_Factorial(N - 1);
      end if;
   end Calcular_Factorial;
   Numero : Integer;
   S : String(1..10);
   Last : Natural;
begin
   Put("Ingrese un número para calcular el factorial: ");
   Get_Line(File => Standard_Input, Item => S, Last => Last);
   Numero := Integer'Value(S(1..Last));
   Put_Line("Factorial de " & Integer'Image(Numero) & ": "
   & Integer'Image(Calcular_Factorial(Numero)));
end Factorial;
```

Figure 8: Ejemplo de factorial calculado en *Ada*

Como podemos observar la separación de bloques mediante los palabras clave `begin` y `end` se utiliza en ambos lenguajes, también dentro de los programas hay una estructura de programa basado en declaración de variables y posteriormente cuerpo del programa. A su vez, se aprecia también una similitud en las estructuras condicionales [11].

4.2 Modula-2

Modula-2 es creado por *Niklaus Wirth*, creador de *Pascal*. Este lenguaje, se hizo con el objetivo de superar a Pascal en todo el tema de gestión y modularización de código. Por lo tanto, *Ada* y *Modula-2* comparten bastante rasgos en este sistema de modularización. Estos utilizan un sistema de packages (aunque en en Modula-2 se utilizan los módulos) [3] para poder ordenar el código.

A su vez, ambos programas presentan similitudes a la hora de la implementación del paradigma orientado a objetos [3].

5 Sistema de tipos

Ada es considerado un lenguaje de **tipos estático** y **tipado estricto**. Es decir, todas las variables a la hora de su declaración se ha de especificar su tipo. A su vez, se hace una comprobación en tiempo de compilación de todas las operaciones.

Ada también lleva un paso adelante, debido a su propósito, en el tipado estricto. La principal característica de su tipado es la ausencia de conversiones implícitas [10].

A su vez, la creación de tipos nuevos a partir de ya existentes, para poder proporcionar esa seguridad en todo los programas, es una de las principales características del lenguaje. A continuación podemos ver una lista de declaraciones que podemos hacer:

```
package Own_Types is
   type Color is (Red, Orange, Yellow, Green, Blue, Indigo, Violet);          -- 1 An enumerated type;
      -- an  ordered set of values; not a synonym for a set of  integer values -- 2 A single line comment
   type Farenheit is digits 7 range -473.0..451.0;                            -- 3 Floating point type
   type Money is delta 0.01 digits 12;                                        -- 4 Financial data type for accounting
   type Quarndex is range -3_000..10_000;                                     -- 5 Integer type; note underbar notation
   type Vector is array(1..100) of Farenheit;                                 -- 6 Constrained array type
   type Color_Mix  is array(Color) of Boolean;                                -- 7 Constrained by Color set
   type Inventory is record                                                   -- 8 A constrained record type
     Description : String(1..80) := (others => ' ');                          -- 9 Initialized string type record component
     Identifier  : Positive;                                                  -- 10 A positive type record component
   end record;                                                                -- 11 End of record scope required by Ada
   type Inventory_Pointer is access all Inventory;                            -- 12 Declaring a pointer type in Ada
   type QData is array(Positive range <>) of Quarndex;                        -- 13 Unconstrained array type
   type Account is tagged record                                              -- 14 See next example: 1.5.3.3
     ID            : String (1..20);                                          -- 15 Uninitialized string type component
     Amount        : Money := 0.0;                                            -- 16 See line 4 of this package
   end record;                                                                -- 17 Required by language
   type Account_Ref is access all Account'Class;                              -- 19 Classwide pointer type for tagged type
end Own_Types;
```

Figure 9: Listado de declaraciones de nuevos tipos a partir de otros ya existentes [8].

Como se puede ver la creación de tipos es una tarea muy similar a otros lenguajes como por ejemplo *Haskell*. Aun así, *Ada* va un paso más para poder asegurar fiabilidad y hace comprobaciones de tipo diferentes aunque 2 tipos provengan del mismo (creación de tipos derivados). A continuación mostramos un ejemplo:

```
  procedure Main is
   type Distance is new Float;
   type Area is new Float;

   D1 : Distance := 2.0;
   D2 : Distance := 3.0;
   A  : Area;
begin
   D1 := D1 + D2;         -- OK
   D1 := D1 + A;     -- NOT OK: incompatible types for "+" operator
   A  := D1 * D2;    -- NOT OK: incompatible types for ":=" assignment
   A  := Area (D1 * D2); -- OK
end Main;
```

Figure 10: Ejemplo donde se muestra los errores relacionados con la creación de tipos derivados [10].

Como podemos ver en la figura anterior, el programa crea dos tipos (`Distance` y `Area`) que provienen del mismo tipo (`Float`) y esto hace que el compilador evalué de manera independiente las operaciones de estos tipos con el resto. Haciendo que no se puedan mezclar operaciones entre diferentes tipos, incluso si provienen del mismo.

Aún y esto, nos podría que interesar tener tipos que no llegan a ser diferentes de otros, y solo un subconjunto de estos. Esta clase de tipo se les llama subtipos y no dejan de ser una variación de los tipos derivados, donde si que se dejan hacer operaciones entre los tipos que comparten el padre. Esto es muy común para el uso de *alias*, es decir esto no deja de ser ayudas al programador para que este no llegue a cometer errores.

```
with Ada.Text_IO;
procedure TypeAliasExample is
    subtype Interest is Float range 0.0 .. 1.0;
    function MoneyToPay (i: Interest; money: Float) return Float is
        one: Float;
    begin
        one := 1.0;
        return money * (one + i);
    end MoneyToPay;
    i: Interest;
begin
  i := 1.1;
  Ada.Text_IO.Put_Line(Float'Image(MoneyToPay(i,100.0)));
end TypeAliasExample;
```

Figure 11: Programa donde se calcula el dinero a pagar dado una cantidad y un interés

Como podemos observar, debido a que el tipo de interés es un porcentaje (asumiremos que como máximo se tiene un interés del 100 %) creamos un tipo que solo acepte este tipo de rango. Pero a su vez, para calcular el dinero que se tiene que devolver en total, necesitamos que este tipo pueda ser utilizado como un `Float`.Haciendo que en este caso este nuevo tipo creado en realidad sea un subtipo.

Debido a que en el programa asignamos a una variable `i: Interest` y le asignamos el valor de 1.1, el programa dará un error de rango. Es decir, gracias a la creación de subtipos, nos ayuda a reducir el número de errores que puede llegar a tener el programa y a su vez, nos da toda las operaciones para poder interactuar con los elementos que tienen el mismo tipo padre.

6 Sistema de ejecución

Ada es un lenguaje compilado. Por lo tanto, es un lenguaje que no se ha desarrollado un sistema de interprete (a diferencia de otros lenguajes como Haskell). Para poder compilar en *Ada* tenemos diferentes compiladores, pero los principales son **GNAT** y *GCC*. La principal diferencia entre estos 2 compiladores, es el alcance de estos. *GNAT* ofrece el proceso de compilación como el de linking, *GCC* en cambio solo ofrece el proceso de compilado, por lo tanto necesitaremos software externo (ej: *GNAT*) para poder generar el ejecutable. A continuación se muestra 2 ejemplos de como compilar en ambos compiladores mencionados.

```
        $ gnatmake -o hello.exe hello.adb
```

Figure 12: Comando para poder compilar un archivo *hello.adb* con el compilador *GNAT* (el ejecutable se nombra *hello.exe*) [4]

```
$ gcc -c hello.adb
$ gnatbind -x hello.ali
$ gnatlink -o hello.exe hello.ali
```

Figure 13: Comando para poder compilar un archivo *hello.adb* con el compilador *GCC* y haciendo el proceso de linking con *GNAT* (el ejecutable se nombra *hello.exe*) [4]

Como podemos observar, ambos compiladores aceptan *flags*. Es decir, podemos obtener cierta configuración a la hora de compilar. Esto es muy beneficioso a la hora de las optimizaciones de compilación. Por ejemplo, gracias al uso de *GCC* podemos utilizar *flags* de optimización que ayuden a mejor el rendimiento del ejecutable.

7 Opinión personal y crítica

Desde mi punto de vista, *Ada* cumple con notoriedad el propósito del lenguaje. Gracias a su tipado estricto y todas las restricciones. Obliga al programador a tener que pensar su código de una manera en que no pueda llegar a tener vulnerabilidades. A su vez, al ser un lenguaje compilado, nos ofrece esa rapidez que necesitamos en sistemas críticos.

Unos inconvenientes que puede tener es su alta curva aprendizaje y actualmente que está teniendo poco uso a nivel general. Lo que llega a provocar una cierta dificultad a la hora de buscar información y recursos para poder entender este lenguaje. Pero podemos entender las bases gracias a la documentación proporcionada por la web oficial de *Ada*.

Por último quiero destacar, las nuevas características que van implementando en las nuevas versiones del lenguaje con el objetivo de aumentar la seguridad en los programas. Algunas de estas técnicas son implementadas por muy pocos lenguajes. Por ejemplo programación contractual basada en pre y post-condiciones que ofrece un grado más de seguridad en todos nuestros programas.

Es decir, este lenguaje está en constante evolución y cada vez ofrece más niveles de fiabilidad en sus programas. Una cosa que todo programador quiere ofrecer.

8 Evaluación de la fuentes

Las fuentes de información utilizadas para este trabajo son principalmente de la página web oficial de *Ada* y sus recursos para aprender el lenguaje. Para datos históricos, puntuales y también para llegar a tener un punto de partida para saber las características del lenguaje se ha utilizado *Wikipedia*, contrastada con las páginas webs oficiales del lenguaje.

Para la sección de Historia se han utilizado tanto la web oficial que explica a rasgos generales todo el lenguaje *Ada* [1], como la web que explica su historia [9]. Por último, se ha utilizado

la web de *Wikipedia* [2] para saber poder saber información sobre *Ada Lovelace*.

Para la sección de paradigmas se ha utilizado como punto de partida *Ada Lovelace* y posteriormente hemos contrastado los paradigmas más importantes con su correspondiente sección en webs oficiales del lenguaje. Concretamente se ha utilizado [5], [6] y [7].

Siguiendo la misma estrategia que la sección anterior. Como punto de partida se utiliza [2] y posteriormente se ha utilizado artículos donde se comparan cada lenguaje mencionado con *Ada* ([11], [3]).

La información sobre el sistema de tipos de Ada se basa casi completamente sobre la web oficial sobre los tipos de Ada [10], complementada con un libro para poder aclarecer el alcance de sus tipos [8].

Por ultimo, para saber el sistema de ejecución. Se basa completamente en la web oficial de Ada donde explica la compilación en ambos compiladores([4]).

Bibliografía y Webgrafía

[1] *About Ada*. URL: https://www.adacore.com/about-ada. Accedido: 23.11.2023.

[2] *ADA (programming lenguage)*. Nov. 12, 2023. URL: https://en.wikipedia.org/wiki/Ada_(programming_language). Accedido: 21.11.2023.

[3] Richard Bielak. "ADA(*) VS. MODULA-2: A VIEW FROM THE TRENCHES". In: *ACM SIGPLAN Notices* (1985). Accedido: 28.11.2023.

[4] *Building Executable Programs with GNAT*. URL: https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/building_executable_programs_with_gnat.html#. Accedido: 1.12.2023.

[5] *Concurrency*. URL: https://learn.adacore.com/courses/Ada_For_The_CPP_Java_Developer/chapters/11_Concurrency.html. Accedido: 26.11.2023.

[6] *Generics*. URL: https://learn.adacore.com/courses/Ada_For_The_CPP_Java_Developer/chapters/09_Generics.html#. Accedido: 26.11.2023.

[7] *Object-oriented programming*. URL: https://learn.adacore.com/courses/intro-to-ada/chapters/object_oriented_programming.html. Accedido: 24.11.2023.

[8] Richard Riehle. *Ada Distilled for Ada 2005*. AdaPower.com, 2009. Accedido: 30.11.2023.

[9] *Timeline of the Ada Programming Language*. URL: https://www.adacore.com/about-ada/timeline-of-ada. Accedido: 21.11.2023.

[10] *Type System*. URL: https://learn.adacore.com/courses/Ada_For_The_CPP_Java_Developer/chapters/05_Type_System.html. Accedido: 30.11.2023.

[11] B. Wichmann. "A Comparison of Pascal and Ada". In: *Comput. J.* 25 (Feb. 1982), pp. 248–252. DOI: 10.1093/comjnl/25.2.248. Accedido: 28.11.2023.