Esta práctica tiene como objetivo la familiarización con el entorno de compilación y el entorno de simulación de Ada.

1 Instalación de paquetes GNAT

GNAT (GNU NYU Ada Translator) es un compilador libre y de código abierto para el lenguaje de programación Ada, desarrollado como parte del proyecto GCC (GNU Compiler Collection). GNAT permite compilar programas en Ada, generando código ejecutable optimizado para diferentes arquitecturas y sistemas operativos. Sus características son:

- Basado en GCC: GNAT forma parte del conjunto de compiladores de GNU y aprovecha todas las optimizaciones y herramientas del ecosistema GCC.
- Compatibilidad con Ada: Soporta las versiones más recientes del estándar Ada (Ada 83, Ada 95, Ada 2005, Ada 2012 y Ada 2022). Además, permite integrar código Ada con otros lenguajes como C y C++.
- Multiplataforma: Está disponible para múltiples sistemas operativos, incluidos Linux, Windows y macOS.

Dependiendo del sistema operativo, GNAT se puede instalar de diferentes maneras.

1.1 Instalación en Linux (Ubuntu/Debian)

1. Abre el terminal.
2. Actualiza la lista de paquetes:

```
$sudo apt update
```

3. Instala el compilador GNAT:

```
$sudo apt install gnat
```

4. Verifica que GNAT se ha instalado correctamente:

```
$gnat --version
```

1.2 Instalación en Windows/macOS

1. Descarga e instala GNAT para Windows desde el sitio web oficial.
2. Sigue las instrucciones del instalador.

2 Lenguaje de programación Ada

2.1 Conceptos básicos de Ada

Ada es un lenguaje de programación estructurado, tipado de manera estática, diseñado para sistemas de alta confiabilidad, como sistemas embebidos, aeronáuticos, de defensa y aplicaciones críticas en general. Ada es un lenguaje que promueve la programación modular y la verificación de sistemas mediante el

uso de un sistema de tipos fuerte. Además, Ada es un lenguaje de programación concurrente, lo que significa que soporta de forma nativa la ejecución de procesos concurrentes (tareas), lo que lo hace adecuado para aplicaciones en tiempo real. Sus características son:

- **Estructurado**: Ada permite que los programas se organicen en módulos (paquetes) y procedimientos, favoreciendo un enfoque modular y la reutilización del código. Esto mejora la legibilidad y mantenibilidad del código.
- **Tipado estático**: En Ada, los tipos de datos de todas las variables y constantes deben ser definidos explícitamente. El compilador verifica que los tipos se utilicen correctamente en el código, lo que ayuda a evitar muchos errores comunes de programación.
- **Concurrente**: Ada es conocido por su soporte de tareas, que permite la ejecución concurrente de múltiples actividades dentro de un programa. Esto es útil en sistemas en tiempo real donde múltiples eventos pueden ocurrir simultáneamente.

2.2 Estructura básica de un programa en Ada

Un programa en Ada se organiza en bloques llamados **procedimientos** o **funciones**. Los procedimientos son bloques de código que realizan una tarea, mientras que las funciones también devuelven un valor. Los procedimientos y funciones pueden ser organizados en **paquetes**, que agrupan código relacionado. La estructura básica de un programa en Ada se organiza de la siguiente forma:

```
with Ada.Text_IO; -- Importa el paquete de entrada/salida
procedure Hola is
begin
    Ada.Text_IO.Put_Line("¡Hola, mundo!");
end Hola;
```

En este ejemplo:

- La declaración `with Ada.Text_IO;` importa el paquete necesario para realizar operaciones de entrada/salida.
- El bloque `procedure Hola is ... end Hola;` define un procedimiento llamado Hola.
- Dentro del procedimiento, se llama a `Ada.Text_IO.Put_Line` para imprimir un mensaje en la consola.

2.3 Tipos de datos en Ada

Ada es un lenguaje de programación de tipado fuerte, lo que significa que se deben definir los tipos de las variables antes de utilizarlas. Algunos de los tipos de datos más comunes en Ada son:

- **Enteros**: Ada tiene tipos de datos para representar números enteros, como `Integer`.
- **Reales**: Para representar números de punto flotante se utiliza el tipo `Float`.
- **Booleanos**: Ada permite trabajar con valores lógicos `True` o `False`.
- **Cadenas de caracteres**: Para trabajar con texto, se utiliza el tipo `String`.
- **Tipos definidos por el usuario**: Ada permite a los usuarios crear sus propios tipos de datos mediante el uso de `type`.

Ejemplo de tipos de datos en Ada:

```
procedure Ejemplo_Tipos is
    A : Integer := 10; -- Variable de tipo entero
    B : Float := 3.14; -- Variable de tipo real
    C : Boolean := True; -- Variable de tipo booleano
    D : String := "Ada"; -- Variable de tipo cadena de caracteres
begin
    -- Código que utiliza las variables
end Ejemplo_Tipos;
```

3 Entorno de compilación y ejecución en Ada

Supongamos que hemos guardado el código del "*Hola Mundo*" en un fichero llamado `hola.adb`. Para compilar y ejecutar con GNAT:

```
$gnatmake hola.adb   # Compila el programa
$./hola              # Ejecuta el binario
```

También se puede compilar en pasos separados:

```
$gnat compile hola.adb   # Genera el archivo objeto hola.o
$gnat bind hola          # Enlaza el archivo objeto
$gnat link hola          # Genera el ejecutable
$./hola                  # Ejecuta el programa
```

Con el primer paso, GNAT genera dos archivos dependiendo de la configuración y el contenido del código. En general, los archivos generados son:

- hola.o → Código objeto compilado (binario intermedio).
- hola.ali → Archivo de información de compilación de Ada (Ada Library Information).

El segundo paso, GNAT genera dos archivos intermedios que se usarán en el siguiente paso. Estos ficheros son ejecutables temporales que se ejecutan en el siguiente paso (gnat link) para comprobar si hay problemas en la vinculación de paquetes antes de generar el ejecutable final. El resultado final, será el ejecutable.

4 Depuración y errores comunes

4.1 Errores comunes y cómo solucionarlos

A la hora de compilar ficheros de Ada, pueden aparecer diversos errores:

- **Errores de sintaxis**: Si olvidas un ; al final de una declaración, el compilador te indicará un error de sintaxis.
- **Errores de tipo**: Ada es un lenguaje fuertemente tipado. Si intentas sumar una cadena con un número, obtendrás un error de tipo.

Para depurar, puedes usar el comando `$gnatmake -g hola.adb` para generar información de depuración.

4.2 Estructuras de Control en Ada

4.2.1 Condicionales

Instrucción if: Permite ejecutar diferentes bloques de código según una condición.
Ejemplo: Determinar si un número es positivo, negativo o cero.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Evaluar_Numero is
   N : Integer;
begin
   Put("Introduce un número: ");
   Get(N);
   if N > 0 then
      Put_Line("El número es positivo.");
   elsif N < 0 then
      Put_Line("El número es negativo.");
   else
      Put_Line("El número es cero.");
   end if;
end Evaluar_Numero;
```

Instrucción case: Ideal cuando hay múltiples valores posibles para una variable.
Ejemplo: Menú de opciones.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Menu is
   Opcion : Character;
begin
   Put_Line("Menú:");
   Put_Line("a) Saludar");
   Put_Line("b) Decir adiós");
   Put_Line("c) Salir");
   Put("Elige una opción: ");
   Get(Opcion);
   case Opcion is
      when 'a' => Put_Line("¡Hola!");
      when 'b' => Put_Line("¡Adiós!");
      when others => Put_Line("Opción no válida.");
   end case;
end Menu;
```

4.2.2 Bucles

Instrucción loop ... exit: Permite ejecutar código repetidamente hasta cumplir la condición.
Ejemplo: Pedir números hasta que el usuario introduzca 0.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Pedir_Numeros is
   N : Integer;
begin
   loop
      Put("Introduce un número (0 para salir): ");
      Get(N);
      exit when N = 0;
      Put_Line("Número insertado: " & Integer'Image(N));
   end loop;
   Put_Line("Fin del programa.");
end Pedir_Numeros;
```

Instrucción while: Ejecuta código mientras se cumpla una condición.
Ejemplo: Contador regresivo.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Cuenta_Regresiva is
   N : Integer := 5;
begin
   while N > 0 loop
      Put_Line("Contando: " & Integer'Image(N));
      N := N - 1;
   end loop;
end Cuenta_Regresiva;
```

Instrucción for: Permite recorrer un rango de valores.
Ejemplo: Imprimir los números del 1 al 5.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Contar is
begin
   for I in 1 .. 5 loop
      Put_Line("Número: " & Integer'Image(I));
   end loop;
end Contar;
```

4.3 Funciones y procedimientos

Una función en Ada devuelve un único valor y se usa para calcular y retornar resultados, mientras que un procedimiento modifica variables externas.

4.3.1 Sintaxis básica de una función

```
function Nombre_Funcion (Parametro1 : Tipo) return Tipo is
begin
   -- Cuerpo de la función
   return Valor;
end Nombre_Funcion;
```

Ejemplo: Función que suma dos números enteros:

```
function Sumar (A, B : Integer) return Integer is
begin
   return A + B;
end Sumar;
```

4.3.2 Declaración y uso de funciones

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Prueba_Funcion is
   function Cuadrado (N : Integer) return Integer is
   begin
      return N * N;
   end Cuadrado;

   Resultado : Integer;
begin
   Resultado := Cuadrado(4);
   Put_Line("El cuadrado de 4 es: " & Integer'Image(Resultado));
end Prueba_Funcion;
```

4.3.3 Diferencia entre funciones y procedimientos

Característica	Función	Procedimiento
Devuelve valor	✅ Sí	× No
Se usa en expresiones	✅ Sí	× No
Modifica variables externas	× No	✅ Sí

Figure 1: Tipos de excepciones

Ejemplo de procedimiento (sin retorno):

```
procedure Mostrar_Mensaje is
begin
   Put_Line("¡Hola, Ada!");
end Mostrar_Mensaje;
```

4.3.4 Funciones recursivas

Una función puede llamarse a sí misma para resolver problemas de forma recursiva.

```
function Factorial (N : Integer) return Integer is
begin
   if N = 0 then
      return 1;
   else
      return N * Factorial(N - 1);
   end if;
end Factorial;
```

4.4 Excepciones en Ada

Una excepción es un mecanismo que permite manejar errores o situaciones inesperadas durante la ejecución de un programa. En Ada, si ocurre un error en tiempo de ejecución, se lanza una excepción que puede ser capturada y tratada para evitar que el programa falle de manera abrupta.

4.4.1 Manejo de las excepciones

Ada proporciona el bloque `begin ... exception ... end;` para capturar y manejar errores. La estructura básica es la siguiente:

```
begin
   -- Código que puede causar un error
exception
   when Tipo_De_Excepción =>
      -- Código para manejar el error
end;
```

4.4.2 Tipos de excepciones comunes

Algunos de los errores más frecuentes en Ada y sus excepciones asociadas son:

Excepción	Ocurre cuando...
`Data_Error`	El usuario ingresa un valor de tipo incorrecto (ej., letras en vez de números).
`Constraint_Error`	Se intenta dividir por cero o se excede un límite permitido (ej., índice fuera de rango).
`Program_Error`	Se produce un fallo interno en el programa (ej., llamadas incorrectas a subprogramas).
`Storage_Error`	Se queda sin memoria en tiempo de ejecución.

Figure 2: Tipos de excepciones

4.4.3 Ejemplo práctico: Captura de errores en entrada de usuario

Supongamos que queremos leer un número entero del usuario y evitar que el programa falle si introduce texto en lugar de un número:

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Excepciones is
   Num : Integer;
begin
   begin
      Put("Introduce un número entero: ");
      Get(Num);
      Put_Line("Número insertado: " & Integer'Image(Num));
   exception
      when Data_Error =>
         Put_Line("Error: Entrada inválida. Introduce un número entero.");
   end;
end Excepciones;
```

5 Ejemplos en Ada

5.1 Operaciones con números (Entrada y salida)

Objetivo: Leer dos números y mostrar su suma, resta, multiplicación y división.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Operaciones is
   A, B : Integer;
begin
   Put("Introduce el primer número: ");
   Get(A);
   Put("Introduce el segundo número: ");
   Get(B);
   Put_Line("Suma: " & Integer'Image(A + B));
   Put_Line("Resta: " & Integer'Image(A - B));
   Put_Line("Multiplicación: " & Integer'Image(A * B));
   Put_Line("División: " & Integer'Image(A / B));
end Operaciones;
```

Tareas:

- Modificar la división para que el resultado sea real.
- Modificarlo para que los números sean reales (Float).
- Capturar errores si el usuario ingresa caracteres no numéricos.

5.2 Control de flujo: Números pares

Objetivo: Mostrar los números pares del 1 al 20 utilizando un bucle.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Numeros_Pares is
begin
   for I in 1 .. 20 loop
      if I mod 2 = 0 then
         Put_Line(Integer'Image(I));
      end if;
   end loop;
end Numeros_Pares;
```

Tareas:

- Modificar el programa para que el usuario elija el rango de números.
- Hacer que el programa imprima los números en orden descendente.

5.3 Uso de registros

Objetivo: Definir un registro Alumno con nombre, edad y calificación y mostrar sus datos.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Registro_Alumno is
   type Alumno is record
      Nombre       : String(1 .. 20);
      Edad         : Integer;
      Calificacion : Float;
   end record;

   Estudiante : Alumno;
begin
   Estudiante := (Nombre => "Juan Perez      ",
                  Edad => 20, Calificacion => 9.5);
   Put_Line("Nombre: " & Estudiante.Nombre);
   Put_Line("Edad: " & Integer'Image(Estudiante.Edad));
   Put_Line("Calificación: " & Float'Image(Estudiante.Calificacion));
end Registro_Alumno;
```

Tareas:

- Permitir que el usuario introduzca los datos en tiempo de ejecución.
- Almacenar varios alumnos en un array y mostrar sus datos.

5.4 Listas y Arrays

Objetivo: Definir y manejar listas o arrays en Ada.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Lista_Numeros is
   type Lista is array (1 .. 5) of Integer;
   Numeros : Lista := (1, 2, 3, 4, 5);
begin
   for I in Numeros'Range loop
      Put_Line(Integer'Image(Numeros(I)));
   end loop;
end Lista_Numeros;
```

Tareas:

- Modificar el programa para que el usuario ingrese los valores del array.
- Implementar una función que calcule la suma de todos los elementos de la lista.

6 Entregables

Tarea 1: Mini Agenda de Contactos
Objetivo: Implementar un pequeño programa que permita almacenar, buscar y mostrar contactos usando registros y arrays.

- Requisitos:
  - Definir un registro Contacto con los campos:
    - Nombre (String de tamaño fijo)
    - Teléfono (String)
    - Email (String)
  - Almacenar hasta 10 contactos en un array.
  - Implementar un menú interactivo con las opciones:
    - Añadir contacto (si hay espacio disponible).
    - Buscar contacto por nombre.
    - Mostrar todos los contactos.
  - Usar procedimientos para estructurar el código.

Código en Ada y una breve descripción de cómo funciona el programa.

Tarea 2: Cálculo de Estadísticas

Objetivo: Escribir un programa en Ada que tome una lista de números ingresados por el usuario y calcule estadísticas básicas.

- Requisitos:
  - Leer N números introducidos por el usuario y almacenarlos en un array.
  - Implementar funciones para calcular:
    - El mínimo y el máximo.
    - La media y la varianza.
  - Mostrar los resultados por pantalla de forma clara.
  - Manejar posibles errores en la entrada (asegurar que los datos ingresados sean números).

Código en Ada y capturas de pantalla con ejemplos de ejecución.

Tarea 3: Juego de Adivinanza

Objetivo: Crear un programa que genere un número aleatorio y permita al usuario adivinarlo.

- Requisitos:
  - Generar un número aleatorio entre 1 y 100.
  - Pedir al usuario que adivine el número.
  - Dar pistas después de cada intento:
    - "El número es mayor" si el usuario ingresó un número menor.
    - "El número es menor" si el usuario ingresó un número mayor.
  - Contar cuántos intentos tomó adivinar el número y mostrar el resultado al final.

Código en Ada y ejemplos de ejecución con distintos intentos.