EL LENGUAJE DE PROGRAMACIÓN ADA

DEPTO. DE SISTEMAS DE INFORMACION
FACULTAD DE CIENCIAS EMPRESARIALES
UNIVERSIDAD DEL BIO-BIO

LENGUAJE DE PROGRAMACION ADA

Extractado por Eduardo Jara de J.G.P. Barnes. *Programming in Ada*.

Contenido

1. Introducción

La historia de Ada comienza en 1974 cuando el Departamento de Defensa de los Estados Unidos (DoD) se percató que estaba gastando demasiado en software. Se llevó a cabo un estudio detallado sobre la distribución de los costos y se descubrió que sobre la mitad de éstos estaba directamente relacionado con sistemas incrustados (embedded)

Se realizó un análisis de los lenguajes utilizados en diferentes áreas. Se descubrió que COBOL era el estandar para el procesamiento de datos y FORTRAN lo era para cálculos científicos y numéricos. Aunque estos lenguajes no eran modernos, el hecho que fueran uniformemente utilizados en sus respectivas áreas evitaba duplicaciones.

La situación con respecto a los sistemas incrustados era diferente. La cantidad de lenguajes utilizados era enorme. No sólo cada unidad militar tenía su lenguaje de alto nivel favorito, sino que usaban varios lenguajes assembler. En resultado era que había gastos innecesarios en compiladores y costos adicionales en entrenamiento y mantención debido a la falta de estandarización.

Se determinó que la única forma de controlar efectivamente los costos en los sistemas incrustados era estandarizar el uso de lenguajes de programación. El primer paso en esta dirección fue la generación de un documento en que se delineaban los requerimientos del lenguaje estandar. La primera versión (Strawman) fue publicada en 1975. Después de recibir comentarios de diversas fuentes el documento fue refinado (Woodenman). En junio de 1976 se produjo una nueva versión (Tinman). Este era un documento más específico e identificaba la funcionalidad que se requería del lenguaje.

En esta etapa se evaluaron varios lenguajes existentes respecto a la especificación Tinman. Como se podría esperar ninguno de éstos satisfacía totalmente los requerimientos; por otro lado la impresión general era que sería necesario crear un nuevo lenguaje basado en conceptos de vanguardia en el área de la programación.

Los lenguajes existentes fueron clasificados en tres categorías:

a) “no apropiados”: Lenguajes obsoletos u orientados a otras áreas que no fueron considerados en las etapas siguientes. Por ejemplo, FORTRAN y CORAL 66.
b) “no inapropiados”: Estos lenguajes tampoco eran satisfactorios, pero tenían algunas características interesantes que podían ser tomadas como “inspiración” para enriquecer el estandar. Por ejemplo, RTL/2 y LIS.
c) “bases recomendadas”: Los lenguajes Pascal, PL/I y Algol 68 fueron considerados como posibles puntos de partida para el diseño del lenguaje final.

En este punto el documento de requerimientos fue revisado y reorganizado (Ironman). Se llamó a propuestas para el diseño del lenguaje. Se recibieron diecisiete propuestas de las cuales se eligieron tres para que “compitieran” en paralelo. Los cuatro elegidos fueron CII Honeywell Bull (verde), Intermetrics (Rojo), Softech (Azul) y SRI International (Amarillo). Los códigos de color se introdujeron para que la comparación se realizara anónimamente.

Los diseños iniciales aparecieron a comienzos de 1978 y fueron analizados por varios grupos alrededor del mundo. El DoD juzgó que los diseños Verde y Rojo eran más promisorios que los Azul y Amarillo y éstos últimos fueron eliminados.

Entonces, el desarrollo entró en una segunda fase y se dio a los desarrolladores un año más para refinar sus diseños. Los requerimientos también fueron mejorados a la luz de la retroalimentación recibida de los diseños iniciales (Steelman).

La elección final del lenguaje fue hecha el 2 de mayo de 1979 cuando el “lenguaje verde” desarrollado en CII Honeywell Bull por un equipo internacional liderado por Jean Ichbiah fue declarado ganador.

Entonces el DoD anunció que el nuevo lenguaje sería conocido como Ada en honor de Augusta Ada Byron, condesa de Lovelace (1815-1852). Ada, hija de Lord Byron, fue la asistente y mecenas de Charles Babbage y trabajó en su “máquina analítica”. En un cierto sentido ella fue la primera programadora de la historia.

Entonces el desarrollo de Ada entró a una tercera etapa, el propósito de la cual fue el que los eventuales usuarios hicieran sus comentarios respecto a que tan conveniente era el lenguaje para sus necesidades. Se continuó con otros estudios de los que se concluyó que Ada era un buen lenguaje, pero que en algunas áreas todavía se requerían algunos refinamientos. Después de esto (en julio de 1980) se publicó la primera versión definitiva del lenguaje y se la propuso a la ANSI (America National Standards Institute) como un estandar.

La estandarización por parte de ANSI tomó unos dos años y se le introdujeron algunos cambios a Ada. El Manual de Referencia del Lenguaje estandar de la ANSI fue finalmente publicado en enero de 1983. A esta versión del lenguaje se le conoce como Ada 83.

Tan pronto como se comenzó a utilizar el lenguaje se iniciaron los estudios para su mejora basada en la experiencia práctica de los usuarios con el lenguaje. A la nueva versión se le denominó Ada 9X, entre las principales mejoras hechas sobre la anterior se cuenta la incorporación de mecanismo de herencia en el manejo de tipos con que contaba Ada 83, el cual, a pesar de ser muy poderoso, al carecer de herencia herencia no se adecuaba al paradigma de Orientación a Objetos. A esta segunda versión de Ada se denomina actualmente como Ada 95.

Ada es un lenguaje grande en la medida que enfrenta la mayoría de los aspectos relevantes a la programación de sistemas prácticos en el mundo real. Por ejemplo, es mucho más grande que Pascal, el que a pesar de sus extensiones realmente sólo es adecuado para propósitos de entrenamiento (para lo que fue diseñado) y para programas pequeños. A continuación se enumeran las principales características de Ada:

a) Legibilidad: se reconoce que los programas profesionales se leen con mayor frecuencia de lo que son escritos. Por lo tanto es importante evitar una notación lacónica como en C, que permite escribir un programa rápidamente, pero que hace casi imposible entenderlo, excepto para el autor al poco tiempo de haberlo escrito.
b) Tipificación fuerte: esto asegura que cada objeto tiene un conjunto claramente definido de posibles valores y previene la confusión entre conceptos lógicamente distintos. Como consecuencia de esto, muchos errores pueden ser detectados en tiempo de compilación, en otros lenguajes esto podría conducir a programas ejecutables, pero incorrectos.
c) Programación en gran escala: se necesitan mecanismos de encapsulación, compilación separada y manejo de bibliotecas para escribir programas portables y mantenibles.
d) Manejo de excepciones: es un hecho que los programas raramente son correctos en un cien por ciento. Por este motivo se hace necesario proveer un medio por el cual los programas puedan ser construidos de forma tal que los errores en una parte de éste no repercutan en las demás.
e) Abstracción de datos: como ya se ha mencionado, se puede lograr mayor portabilidad y mantenibilidad si los detalles de la representación de los datos puede ser separada de la especificación de las operaciones lógicas sobre los datos.
f) Tareas: en muchos casos es importante que el programa sea concebido como una serie de actividades paralelas en lugar de una secuencia simple de acciones. Al entregar estas facilidades dentro del lenguaje y no a través de llamadas a un sistema operativo se logra una mayor portabilidad y mantenibilidad.
g) Unidades genéricas: es muchos casos la parte lógica de un programa es independiente de los tipos de valores que son manipulados. Por lo tanto se requiere de un mecanismo para la creación de partes lógicamente relacionadas a partir de un prototipo único. Esto es especialmente útil para la creación de bibliotecas.

2. Descripción general

Uno de los aspectos más importantes en la programación es el reuso de partes de programas existentes para que el esfuerzo de generar nuevo código sea mínimo. Así el concepto de biblioteca de programas emerge en forma natural y un aspecto importante de un lenguaje de programación es su habilidad para expresar el cómo reusar itemes de una biblioteca.

Ada reconoce esta necesidad e introduce el concepto de unidades de biblioteca (library units). Un programa Ada completo es concebido como un programa principal (en sí mismo una unidad de biblioteca) que solicita los servicios de otras unidades.

Supongamos que queremos escribir un programa que imprime la raíz cuadrada de un cierto número, por ejemplo, 2.5. Podría esperarse que estén disponibles bibliotecas que nos provean servicios para calcular raíces cuadradas y entregar datos. De este modo, nuestro trabajo se reduciría a escribir un programa principal que haga uso de estos servicios en el modo que deseemos.

Supondremos que en nuestra biblioteca existe una función de nombre SQRT que permite calcular raíces cuadradas, y que, además, nuestra biblioteca cuenta con un paquete (package) llamado SIMPLE_IO que contiene servicios simples de entrada-salida (lectura y escritura de números, escrituras de cadenas de caracteres, etc.). Nuestro programa sería:

```
with SQRT, SIMPLE_IO;
procedure PRINT_ROOT is
   use SIMPE_IO;
begin
   PUT(SQRT(2.5));
end PRINT_ROOT;
```

El programa está escrito como un procedimiento llamado PRINT_ROOT precedido por una cláusula **with** donde se dan los nombre de las unidades de biblioteca que se desea usar. El cuerpo del procedimiento contiene una única instrucción:

```
PUT(SQRT(2.5));
```

Al escribir

```
use SIMPLE_IO;
```

se obtiene acceso inmediato a los servicios del paquete SIMPLE_IO. Si se hubiese omitido la cláusula **use** hubiese sido necesario usar la "notación punto".

```
SIMPLE_IO.PUT(SQRT(2.5));
```

para indicar dónde debe buscarse el procedimiento PUT.

Podemos hacer nuestro programa más útil haciendo que lea el número cuya raíz cuadrada se desea obtener. El programa quedaría:

```
with SQRT, SIMPLE_IO;
procedure PRINT_ROOT is
   use SIMPE_IO;
   X : FLOAT;
begin
   GET(X);
   PUT(SQRT(X));
end PRINT_ROOT;
```

La estructura general del procedimiento es clara: entre **is** y **begin** podemos escribir declaraciones, y entre **begin** y **end** escribimos instrucciones (operaciones). En términos generales podemos decir que las declaraciones presentan las entidades que queremos manipular y las instrucciones indican la secuencia de acciones a realizar.

Notemos algunos detalles:

- Todas las declaraciones e instrucciones terminan con un punto y coma a diferencia de otros lenguajes como Algol y Pascal donde los punto y coma son utilizados como separadores en lugar de terminadores.
- El programa contiene varios identificadores tales como **procedure** , PUT y X. Los identificadores se dividen en dos grandes grupos. Unos pocos (63 en Ada 83) como **procedure** e **is** , que son utilizados para indicar la estructura del programa; son palabras reservadas y no pueden ser usados para otros propósitos. Los otros tales como PUT y X pueden ser usados para cualquier propósito que se desee. Algunos de ellos (en nuestro ejemplo FLOAT) tienen un significado predefinido, pero podrían ser usados para otros propósitos, aunque se corre el riesgo de confusión en el código.

Finalmente observemos que el nombre del procedimiento, PRINT_ROOT, se repite entre el **end** final y el punto y coma. Esto es opcional, pero se recomienda para aclarar la estructura general, el cual es obvio en un ejemplo tan pequeño.

Nuestro ejemplo es demasiado simple, sería más útil si se lee una serie de números y se imprime cada respuesta en una línea separada. Arbitrariamente definiremos que el proceso terminará al ingresarse un valor igual a cero.

```
with SQRT, SIMPLE_IO;
procedure PRINT_ROOTS is
    use SIMPLE_IO;
    X : FLOAT;
begin
    PUT("Roots of various numbers");
    NEW_LINE(2);
    loop
        GET(X);
        exit when X = 0.0;
        PUT("Root of");
        PUT(X);
        PUT("is");
        if X < 0.0 then
                PUT("not calculable");
        else
                PUT(SQRT(X));
        end if;
        NEW_LINE;
   end loop;
    NEW_LINE;
    PUT("Program finished");
    NEW_LINE;
end PRINT_ROOTS;
```

La salida del programa ha sido mejorada mediante la llamada a los procedimientos NEW_LINE y PUT incluidos en el paquete SIMPLE_IO. El parámetro numérico entero del procedimiento NEW_LINE indica cuantas líneas se deberá escribir, si no se entrega el parámetro se asume un valor igual a 1 (uno). También hay una llamada a PUT con una cadena como argumento. Este es de hecho un procedimiento diferente al que escribe el número X. El compilador diferencia cual corresponde en cada caso de acuerdo al tipo (y/o cantidad) de parámetros utilizado.

En Ada se debe observar reglas estrictas para cerrar estructuras; **loop** es cerrado con **end loop** e **if** con **enf if**. Todas las estructuras de control de Ada tienen sus cierres en lugar de la forma abierta de Pascal que puede conducir a errores.

Veamos ahora la posible estructura de la función SQRT y el paquete SIMPLE_IO que en nuestro ejemplo se han supuesto como existentes.

La función SQRT tendrá una estructura similar a nuestro programa, la mayor diferencia radica en la existencia de parámetros.

```
function SQRT (F:FLOAT) return FLOAT is
        R: FLOAT;
begin
        --- calcular la raíz cuadrada de F y guardarla en R
        return R;
end SQRT;
```

El paquete SIMPLE_IO consta de dos partes, la especificación que describe su interfaz con el mundo externo (es decir, las otras unidades de biblioteca que hacen uso de el paquete) y el cuerpo (**body**) que contiene los detalles de cómo han sido implementados los componentes del paquete. Si el paquete contiene sólo los procedimientos que hemos usado en nuestro ejemplo, su especificación sería:

```
package SIMPLE_IO is
        procedure GET(F: out FLOAT);
        procedure PUT(F: in FLOAT);
        procedure  PUT(S: in STRING);
        procedure NEW_LINE(N: in INTEGER:=1);
end SIMPLE_IO;
```

El parámetro de GET es un parámetro **out** (salida) puesto que el efecto de llamar a este procedimiento en GET(X) es entregar un valor "desde dentro" del procedimiento por medio del parámetro

real X (**actual parameter** X). Los demás parámetros son **in** puesto que sirven para ingresar datos a los procedimientos.

Sólo una parte de los procedimientos está presente en la especificación del paquete; esta parte se conoce como la "especificación de los procedimientos" donde sólo se entrega la información que permite llamar a los procedimientos. Es decir, para que nuestro programa PRINT_ROOTS use el procedimiento GET sólo necesita saber el nombre de éste y sus parámetros, los detalles de implementación en esta etapa no importan. Esto significa que para compilar el programa PRINT_ROOTS sólo es necesario que esté presente la declaración del paquete SIMPLE_IO. Lógicamente, para poder ejecutar el programa se necesitará la implementación de todos los procedimientos.

En nuestro ejemplo podemos ver dos especificaciones superpuestas de PUT, una con un parámetro de tipo FLOAT y otro con un parámetro de tipo STRING. Es importante recalcar que estos son dos procedimientos diferentes, el compilador Ada discriminará de acuerdo a los parámetros de cada uno de ellos. Finalmente, notemos como se asigna un valor por omisión al parámetro del procedimiento NEW_LINE.

El cuerpo del paquete (package body) SIMPLE_IO contendrá la totalidad de los cuerpos de los procedimientos especificados, además de otros elementos (variables, estructuras de datos, otras funciones o procedimientos) que se requieran para su implementación y que quedan de una forma natural escondidas de los usuarios externos. Esto quiere decir que los usuarios del paquete (otras unidades de biblioteca: funciones, procedimientos, paquetes, tareas) sólo "conocen" lo indicado en la especificación del paquete. En términos generales el cuerpo del paquete tendría la siguiente estructura:

```
with INPUT_OUTPUT;
package body SIMPLE_IO is
        . . .
        procedure GET (F: out FLOAT) is
                . . .
        begin
                . . .
        end GET;
        -- otros procedimientos parecidos
end SIMPLE_IO
```

La cláusula **with** muestra que la implementación (pero, en este caso, no la especificación) de los procedimientos en SIMPLE_IO requiere de un hipotético paquete más general llamado INPUT_OUTPUT. Notemos, además, que en el cuerpo de GET se repite la especificación del procedimiento que fue entregada en la especificación del paquete.

Es importante mencionar el paquete especial STANDARD. Este es un paquete que existe en toda implementación del lenguaje y contiene las declaraciones de todos los identificadores predefinidos tales como FLOAT y NUMERIC_ERROR. Se asume el acceso automático a este paquete y por lo tanto no es necesario dar su nombre en una cláusula **with**.

En resumen, un programa Ada es concebido como un conjunto de componentes (procedimientos, funciones, paquetes, tareas) que se proveen servicios mutuamente.

Ejercicio: Seguramente la función SQRT no estará directamente disponible en una biblioteca, sino que será parte de un paquete junto con otras funciones matemáticas. Supongamos que dicho paquete tiene por nombre SIMPLE_MATHS y los otras funciones son LOG, EXP, SIN y COS. Se pide que escriba la especificación de dicho paquete (utilice la especificación de SIMPLE_IO como modelo) ¿Qué cambios habría que hacerle al programa PRINT_ROOTS?

3. Estilo léxico

Ciertamente no es muy agradable comenzar el estudio de un lenguaje con un tema tan árido como son los detalles de la construcción de cosas tales como identificadores y números, sin embargo ello es esencial para un conocimiento acabado de un lenguaje y, obviamente, para la correcta construcción de programas.

3.1. Elementos léxicos

Un programa Ada se escribe como una secuencia de líneas de texto que contienen los siguientes caracteres:

- alfabeto a-z y A-Z
- dígitos 0-9
- otros caracteres " # & ´ ( ) * + , - / : ; < = > _ |
- el carácter blanco

Se debe tener presente que los siguientes delimitadores compuestos no deben contener espacios:

- => usado en when, cases, etc.
- .. usado para rangos
- ** para exponenciación
- := para asignación
- /= no igual
- >= mayor o igual
- <= menor o igual
- <> para arreglos

3. 2. Identificadores

Un identificador se define de la siguiente manera:

identifier ::= letter {[underline] letter_or_digit}
letter_or_digit ::= letter | digit
letter ::= upper_case_letter | lower_case_letter

Esto quiere decir que un identificador consiste en una letra seguida (posiblemente) de una o más letras o dígitos con subrayados aislados. Se pueden usar letras mayúsculas y minúsculas, las que no son tomadas como diferenciadores de identificadores, por ejemplo:

Sueldo_Base y SUELDO_BASE

son el mismo identificador. A pesar de esta libertad para escribir los identificadores, se recomienda, para una mejor legibilidad usar minúsculas para las palabras reservadas y mayúsculas para los demás identificadores.

Ada no impone un límite en el largo del identificador, sin embargo, puede haber limitaciones impuestas por implementaciones específicas. De esta manara se estimula el uso de nombres de variables autodocumentados como SUELDO_BASE en lugar del poco significativo S.

Ejercicio: Indique cuáles de los siguientes identificadores son incorrectos y por qué

a) Ada
b) fish&chips
c) RATE-OF-FLOW
d) UMO164G
e) TIME_ _LAG
f) 77E2
g) X_
h) tax rate
i) goto

3.3. Números

Los números pueden ser enteros o reales. La mayor diferencia es que los reales siempre contienen un punto decimal y los enteros no. Es ilegal usar un entero donde el contexto indica que debe usarse un real y viceversa. Entonces

AGE: INTEGER := 43.9; y WEIGHT: REAL := 150;

son expresiones incorrectas.

La forma más simple de un entero es una secuencia de dígitos, y la de un real es una secuencia de dígitos con un punto decimal. Nótese que debe haber al menos un dígito a cada lado del punto decimal.

A diferencia de otros lenguajes en Ada tanto los enteros como los reales pueden tener exponente, el cual se indica con una letra E (puede ser minúscula) seguida de un entero con o sin signo. El exponente no puede ser negativo en el caso de un entero (si así fuera el resultado podría no ser un entero). Por ejemplo el real 98.4 podría ser escrito con exponente de las siguientes formas:

9.84E1 98.4e0 984.0e-1 0.984E+2

pero 984e-1 sería incorrecto.

En forma análoga, el entero 1900 podría escribirse con exponente como:

19E2 190e+1 1900E+0

pero 19000 e-1 sería incorrecto.

Ejercicio: Indique cuáles de las siguientes secuencias de dígitos son reales o enteros válidos.

a) 38.6
b) .5
c) 32e2
d) 32e-2
e) E+6
f) 27.4e_2

3.4. Comentarios

Un comentario en Ada es cualquier texto ubicado a la derecha de dos guiones (seguidos), por ejemplo:

```
-- Este es un comentario

PUT(SQRT(2.5));  -- Este es otro comentario
```

El comentario se extiende hasta el final de la línea.

4. Tipos escalares

En este capítulo se hechan las bases de los aspectos a pequeñas escala de Ada. Se comienza con la declaración de objetos, la asignación de valores a ellos y las idea de rango de validez y visibilidad.Se introducen los importantes conceptos de tipo, subtipos y restricciones.

4.1. Declaración de objetos y asignaciones

Los valores (datos) pueden almacenarse en objetos que son de un cierto tipo (lo que se indica al momento de su declaración). Estos objetos pueden ser variables o constantes. Una variable se declara mediante su nombre (un identificador), su tipo y (posiblemente) un valor inicial. Por ejemplo:

```
I : INTEGER;          -- variable de nombre I y de tipo entero
P: INTEGER := 38; -- variable de nombre P, de tipo entero y valor inicial 38
```

Es posible declarar varias variables simultáneamente separándolas por comas. Por ejemplo:

```
I,J,K : INTEGER;
P,Q,R: INTEGER:= 38;
```

Si se declara una variable sin un valor inicial es necesario tener cuidado de no usarla con un valor indefinido. Si un programa usa un valor indefinido de una variable no inicializada, su comportamiento puede se impredecible. En este caso el programa debería considerarse estrictamente erróneo, sin embargo tanto el compilador como el sistema de run-time podrían no ser capaces de detectar el problema.

La manera más simple de asignar un valor a una variable es mediante el comando de asignación (:=). Por ejemplo:

```
I:= 36;
P:=Q + R;
```

Existe una gran similitud entre una declaración con valor inicial y el comando de asignación. Ambos usan := antes de la expresión, la cual puede ser de cualquier nivel de complejidad. Una diferencia importante es que es posible inicializar varias variables simultáneamente, pero no es posible usar un único comando de asignación para varias variables. Esto parece poco práctico, sin embargo la necesidad de asignar un mismo valor a varias variables generalmente sólo surge al momento de inicializarlas.

Una constante se declara de forma similar a una variable, escribiendo la palabra **constant** después de los dos puntos. Lógicamente, una constante debe ser inicializada al ser declarada, de otro modo pierde su sentido. Por ejemplo:

```
PI : constant REAL :=3.1415926536;
```

Ejercicio: Indique los errores de las siguientes declaraciones y comandos:

a) **var** I:INTEGER;
b) G:**constant** :=981;
c) P,Q:**constant** INTEGER;
d) P:=Q:=7;
e) MN:**constant** INTEGER:=M*N;
f) 2PI:**constant**:= 2.0*PI;

4.2. Bloques y ámbito de validez (scope)

Ada hace una clara distinción entre la declaraciones de nuevos identificadores y los instrucciones que los usan. Es obvio que las primeras deben preceder a los últimos. Un bloque (**block**) es la estructura más simple que incluye declaraciones y comandos.

Un bloque comienza con la palabra reservada **declare**, algunas declaraciones, **begin**, algunas instrucciones y concluye con la palabra reservada **end** y un punto y coma. Por ejemplo:

```
declare
    I: INTEGER:=0;
begin
    I:=I+1;
end;
```

Un bloque es un caso particular de una instrucción, por lo tanto una de sus instrucciones internas puede ser otro bloque.

Cuando se ejecuta una instrucción "bloque" se prepara su parte declarativa (entre **declare** y **begin**) y luego se ejecutan las instrucciones (entre **begin** y **end**). Nótese la terminología: se preparan las declaraciones y se ejecutan las instrucciones. La preparación de una declaración consiste en "crear" el objeto declarado y asignarle (si corresponde) un valor inicial. Al llegar al final (**end**) del bloque todas las cosas declaradas en él automáticamente dejan de existir.

Un punto importante es que los objetos usados para la inicialización de otros objetos deben, lógicamente, existir. Es decir, toda declaración debe preceder a su uso. Por ejemplo:

```
declare
    I: INTEGER :=0;
    K: INTEGER:= I;
begin
```

está permitido, pero

```
declare
    K: INTEGER:= I;
    I: INTEGER:= 0;
begin
```

en general no. (¿En qué situación esto estaría correcto, aunque con un significado diferente?)

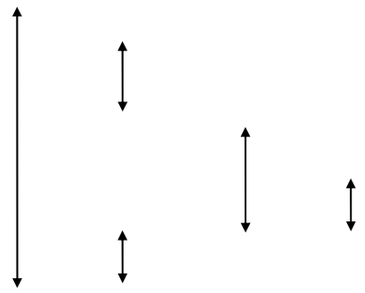
Al igual que otros lenguajes Ada maneja la idea de ocultamiento (hiding). Consideremos el siguiente bloque

```
declare
    I,J: INTEGER;
begin
     . . .          -- aquí es válida la variable I externa
    declare
       I: INTEGER;
    begin
        . . .       -- aquí es válida la variable I interna
    end;
    . . .           -- aquí es válida la variable I externa
end;
```

La declaración interna de I no hace que la externa desaparezca, sino que ésta se torna temporalmente invisible. Al terminar el bloque interno, la variable I interna deja de existir y la externa vuelve a estar visible.

Hay que distinguir entre el ámbito de validez (scope) y la visibilidad de un objeto. Por ejemplo, en el caso de un bloque el ámbito de validez de una variable (o constante) se extiende desde el punto de su declaración hasta el final (end) del bloque. Sin embargo, una variable (o constante) no es visible en su propia declaración ni en ningún bloque interno donde su nombre sea utilizado para la declaración de otro objeto. Los ámbitos de validez y visibilidad de ilustran en el siguiente ejemplo

```
declare
      I: INTEGER:=0;
begin
     declare
          K: INTEGER:= I;
          I: INTEGER:= 0;
     begin
          . . .
     end;
     . . .
end;
```



scope	visibilidad	scope	visibilidad
I externa	I externa	I interna	I interna

Para inicializar K se utiliza el valor de I externa, puesto que la declaración de K precede a la de la variable I interna. Vemos pues que el código

```
K: INTEGER:= I;
I: INTEGER:= 0;
```

puede o no ser válido, depende del contexto.

Ejercicio: Indique qué errores hay en la siguiente sección de código.

```
declare
     I: INTEGER:= 7;
     J,K: INTEGER
begin
     J:= I+K;
     declare
          P: INTEGER = I;
          I,J: INTEGER;
     begin
          I:= P+Q;
          J:= P -Q;
          K:= I* J;
     end;
     PUT(K);
end;
```

4.3. Tipos

Un tipo queda totalmente caracterizado por un conjunto de valores y un conjunto de operaciones. En el caso del tipo de datos primitivo INTEGER, el conjunto de valores está representado por

. . . -3, -2, -1, 0, 1, 2, 3 . . .

y las operaciones por

+, -, *, etc.

Se asume que los tipos primitivos del lenguaje están definidos en el paquete STANDARD.

Los valores de un tipo no pueden ser asignados a variables de otro tipo. Esta una regla fundamental del manejo estricto de tipos.

La declaración de tipos se realiza con una sintaxis distinta a la declaración de variables y constantes para enfatizar la diferencia conceptual existente. La declaración se realiza mediante la palabra reservada **type**, el identificador asociado con el tipo, la palabra reservada **is** y la definición del tipo seguida de un punto y coma. Por lo tanto, el paquete STANDARD debería contener declaraciones tales como

type INTEGER **is .** . ;

La definición entre **is** y **;** de alguna manera entrega el conjunto de valores asociados al tipo. Por ejemplo:

type COLOUR **is** (RED, AMBER, GREEN);

Se ha definido un nuevo tipo denominado COLOUR, al que se asocia un conjunto de 3 valores: RED, AMBER y GREEN, es decir, las variables de tipo COLOUR sólo podrán "almacenar" alguno de dichos valores. Por ejemplo:

```
C: COLOUR;
D: COLOUR:=RED;
DEFAULT: constant COLOUR:=GREEN;
```

son declaraciones válidas.
Debido al manejo estricto de tipos no podemos "mezclar" colores y enteros, por ejemplo:

```
I: INTEGER;
C:COLOUR;
   . . .
I := C;
```

es incorrecto. Comparemos esto con la "filosofía" del leguaje C (Keninghan, B. Ritchie D. El Lenguaje de Programación C).
Una enumeración es una lista de valores enteros constantes, como en

```
enum boolean  {NO, YES};
```

El primer nombre en un **enum** tiene un valor 0, el siguiente 1, y así sucesivamente, a menos que sean especificados valores explícitos. Los valores no especificados continúan la progresión a partir del último valor que sí lo fue, como en el segundo de esos ejemplos:

```
enum escapes  { BELL = ´\a´, RETROCESO = ´\b´, TAB = ´\t´,
                NVALIN = ´\n´, VTAB = ´\v´, RETURN = ´\r´};

enum months { ENE = 1, FEB, MAR, ABR, MAY, JUN,
              JUL, AGO, SEP, OCT, NOV, DIC};
```

Las enumeraciones proporcionan una manera conveniente de asociar valores constantes con nombres, son una alternativa a **#define**. Aunque las variables de tipos **enum** pueden ser declaradas, los compiladores no necesitan revisar que lo que se va a almacenar en tal variable es un valor válido para la enumeración. No obstante, las variables de enumeración ofrecen la oportunidad de revisarlas y tal cosa es a menudo mejor que **#define**. Además, un depurador puede ser capaz de imprimir los valores de variables de enumeración en su forma simbólica.
Como puede verse, en Ada existen mecanismos totalmente diferentes para definir constantes y para definir tipos de enumeración, en cambio en C los tipos **enum** sirven para ambos acciones. Además, lo dicho respecto a tipos **enum** en C es un claro ejemplo de la estilo "relajado" de manejo de tipos de este lenguaje.

4.4. Subtipos

Un subtipo, como su nombre lo indica, es un subconjunto de los valores de otro tipo, al que se le denomina **tipo base**. Dicho subconjunto se define mediante una **restricción**. Si embargo no hay forma de restringir el conjunto de operaciones de tipo base.
Por ejemplo, supongamos que queremos manipular días; sabemos que los días de un mes están en el rango de 1 a 31, entonces el subtipo (del tipo INTEGER) sería

```
subtype DAY_NUMBER is INTEGER range 1 .. 31;
```

Posteriormente podemos declarar variables y constantes usando el identificador de un subtipo exactamente igual a como lo hacemos con un identificador de tipo. Por ejemplo, al declarar

```
D: DAY_NUMBER;
```

estamos asegurando que la variable D pude tomar sólo valores enteros en el rango de 1 a 31. En tiempo de compilación podrían detectarse errores como

```
D := 32;
```

Además, el compilador agregará chequeos de tiempo de ejecución (run time checks) para verificar que se cumpla la restricción, si fallase un chequeo se indicaría un excepción CONSTRAIN_ERROR.
Es importante notar que la declaración de un subtipo no introduce un nuevo tipo. Un objeto como D es de tipo entero y por lo tanto el siguiente código es totalmente legal

```
D: DAY_NUMBER;
I: INTEGER;
  . . .
D:=I;
```

Lógicamente, durante la ejecución, el valor de I podría salirse del rango 1 a 31. En este caso se indicaría CONSTRAIN_ERROR. Pero una asignación en el sentido inverso

```
I := D;
```

siempre funcionará correctamente.

No es necesario declarar un nuevo tipo para imponer una restricción. Por ejemplo, mediante

```
D: INTEGER range 1 .. 31;
```

se ha declara una variable de tipo entero junto con una restricción.

Ejercicio:
1. ¿Cuándo tendrá sentido definir un tipo con una cierta restricción y cuándo una variable?
2. ¿Qué diferencia semántica habrá entre las siguientes declaraciones?
   a) **subtype** DAY_NUMBER **is** INTEGER **range** 1 .. 31;
   b) **type** DAY_NUMBER **is range** 1 .. 31;

4.5. Tipos numéricos simples

La complejidad de los problemas de análisis numérico (estimación de errores y otros) se refleja en los tipos de datos numéricos de Ada. Sin embargo, aquí expondremos sólo los aspectos más simples de los tipos entero (INTEGER) y real (REAL), para usarlos en nuestros ejemplos.

El valor mínimo del tipo INTEGER está dado por INTEGER´FIRST y el máximo por INTEGER´LAST. Estos son dos ejemplos de atributos, los cuales se denotan mediante un apóstrofe seguido de un identificador.

El valor de INTEGER´FIRST dependerá de la implementación y siempre será negativo. En una máquina de 16 bits (con complemento a dos) se tendrá

```
INTEGER´FIRST = -32768
INTEGER´LAST = +32767
```

Por supuesto que para lograr una mejor portabilidad debemos usar los atributos FIRST y LAST en lugar de los valores -32768 y +32767. Dos subtipos muy útiles son

```
subtype NATURAL is INTEGER range 0 .. INTEGER´LAST;
subtype POSITIVE is INTEGER range  1 .. INTEGER´LAST;
```

Los atributos FIRST y LAST se aplican automáticamente a los subtipos, por ejemplo

```
POSITIVE´FIRST = 1
NATURAL´LAST = INTEGER´LAST
```

También es posible hacer restricciones sobre los reales. Además, los atributos FIRST y LAST también son aplicables.

A continuación se resumen algunas de las operaciones predefinidas para los tipos entero y real más importantes.

1) +, - Estos son tanto operadores unarios como binarios. En el primer caso el operando puede ser entero o real; el resultado será del mismo tipo. El operador unario + en realidad no hace nada, pero el operador unario - cambia el signo del operando. Cuando se usan como operadores binarios, ambos operandos deben ser del mismo tipo y el resultado será del tipo que corresponda. Su acciones son la adición y sustracción normales.
2) * Multiplicación; ambos operandos deben ser enteros o reales; el resultado será del tipo correspondiente.
3) / División; ambos operandos deben ser del mismo tipo. La división entero trunca el resultado.
4) **rem** Resto; los operandos deben ser enteros y el resultado también lo es. Entrega el resto de la división.
5) **mod** Módulo; ambos operandos deben ser enteros y el resultado también lo es. Corresponde a la operación matemática módulo.
6) **abs** Valor absoluto; es un operador unario y el operador debe ser entero o real. El resultado es del mismo tipo y corresponde al valor absoluto del operando.
7) ** Exponenciación; eleva el primer operando a la potencia indicada en el segundo. Si el primer operando es un entero, entonces el segundo debe ser un entero positivo o cero. Si el primer operando es un real, entonces el segundo puede ser cualquier entero. El resultado es del mismo tipo del primer operando.

Además, pueden realizarse las operaciones =, /=, <, <=, > y >= para obtener resultados de tipo booleano TRUE o FALSE. También en este caso los operadores deben ser del mismo tipo.

A pesar que estas operaciones son bastante sencillas es necesario hacer algunas observaciones.

La regla general es que no se puede mezclar valores de distinto tipo en una expresión aritmética. Por ejemplo, no se puede sumar un entero a un real. El cambio de un valor INTEGER a uno REAL o viceversa puede hacerse usando el nombre del tipo (o subtipo) de dato que se necesita como si fuera el nombre de una función. Por ejemplo, si tenemos

```
I: INTEGER:=3;
R:REAL:=5.6;
```

no podemos escribir

```
I + R
```

pero si es válida la expresión

```
REAL(I) + R
```

que usa la “suma real” para entregar 8.6, o

```
I + INTEGER(R)
```

que entrega 9, al hacer uso de la “suma entera”.

La conversión de real a entero siempre entrega un valor redondeado, por ejemplo:

1.4 entrega 1
1.6 entrega 2

Dependiendo de la implementación los valores intermedios (como 1.5) serán redondeados hacia arriba o hacia abajo.

Finalmente, hagamos algunas observaciones respecto al operador **. Para un exponente positivo (es decir, un entero positivo) esta operación corresponde a una multiplicación repetida. Por ejemplo:

```
  3**4 = 3*3*3*3 = 81
3.0**4 = 3.0*3.0*3.0*3.0* = 81.0
```

Cuando el segundo operando es cero, el resultado será obviamente uno

```
  3**0 = 1
3.0**0 = 1.0
```

El exponente no puede ser negativo si la base es entera, ya que el resultado podría no ser entero. Pero si la base es real este operador entrega el correspondiente recíproco

```
3.0**(-4) = 1.0/81.0 = 0.0123456780123…
```

Como es usual existen diferentes niveles de precedencia entre los operadores en una expresión, y, además, la precedencia natural puede ser alterada con el uso de paréntesis. Los operadores de igual jerarquía son aplicados de izquierda a derecha. La jerarquía de operados en Ada es la siguiente

```
= /= < <= > >=
+  -              (binarios)
+ -               (unarios)
/ * mod rem
abs **
```

Por ejemplo:

```
A/B*C          significa        (A/B)*C
A+B*C+D        significa        A+(B*C)+D
A*B+C*D        significa        (A*B)+(C*D)
A*B**C         significa        A*(B**C)
```

Como ya se indicó, por omisión los operadores de un mismo nivel se aplican de izquierda a derecha, sin embargo, no está permitido escribir varios operadores de exponenciación sin el uso de paréntesis. Por esto la expresión

```
A**B**C
```

es incorrecta y debe escribirse

```
(A**B)**C        o          A**(B**C)
```

De esta manera se evita el riesgo de que accidentalmente se escriba una expresión sintácticamente correcta, pero semánticamente errónea.

Es necesario tener cuidado con los operadores unarios, por ejemplo:

```
-A**B     significa     -(A**B)     y no     (-A)**B
```

además, las expresiones

```
A**-B   y  A*-B
```

son ilegales. Es necesario, en estos casos, utilizar paréntesis para definir claramente lo que se desea calcular.

Ejecicio:

1. Evalúe las siguientes expresiones:
   I:INTEGER:=7;
   J:INTEGER:=-5;
   K:INTEGER:=3;

a) I*J*K
b) I/J*K
c) I/J/K
d) J+2 rem I
e) K**K**K

2. Escriba las siguientes expresiones matemáticas en Ada. Defina identificadores apropiados.
   a) $b^2 - 4ac$
   b) πr^3

4.6. Tipos de enumeración

Primero mostremos algunos ejemplos de tipos de enumeración:

type COLOUR **is** (RED,AMBER,GREEN);
type DAY **is** (MON, TUE, WED, THU, FRI, SAT, SUN);
type STONE **is** (BERYL, QUARTZ);
type SOLO **is** (ALONE);

No existe un límite superior para la cantidad de valores de un tipo de enumeración, pero debe haber al menos uno.

Las restricciones de los tipos enumerados son semejantes a las restricciones sobre los enteros. Por ejemplo:

subtype WEEKDAY **is** DAY **range** MON .. FRI;
D: WEEKDAY;

Si el límite superior es inferior al límite inferior se obtendrá un rango vacío. Por ejemplo:

subtype COLOURLESS **is** COLOUR **range** AMBER .. RED;

Los atributos FIRST y LAST también son aplicables a los tipos de enumeración, entonces

COLOUR´FIRST = RED
WEEKDAY´LAST = FRI

Otros atributos predefinidos entregan el sucesor (SUCC) y el predecesor (PRED) de un cierto valor de un tipo de enumeración. Por ejemplo:

COLOUR´SUCC(AMBER) = GREEN
STONE´SUCC(BERYL) = QUARTZ
DAY´PRED(FRI) = THU

Lógicamente, lo indicado entre paréntesis es cualquier expresión válida del tipo correspondiente. Si se intenta obtener el antecesor (sucesor) del primer (último) valor el sistema entregará una excepción CONSTRAIN_ERROR.

Otro atributo predefinido es POS, el cual entrega la posición del valor dentro de la declaración del tipo (empezando desde cero). El atributo contrario a POS es VAL, que entrega el valor asociado a una cierta posición. Por ejemplo:

COLOUR´POS(RED) = 0
COLOUR´POS(AMBER) = 1
COLOUR´POS(GREEN) = 2
COLOUR´VAL(0) = RED
DAY´VAL(6) = SUN
SOLO´VAL(1) -- CONSTRAIN_ERROR

Ada incluye los atributos POS y VAL, puesto que podrían ser necesarios en algunas situaciones, pero no se recomienda su uso pues no es una buena práctica de programación. Es mejor pensar en términos de valores de enumeración que en números.

Puesto que se establece un orden entre los valores de un tipo de enumeración , es posible aplicar los operadores =, /=, <, <=, > y >=. Por ejemplo:

RED < GREEN es TRUE
WED >= THU es FALSE

Ejercicio:
a) Evalúe
1) DAY´SUCC(WEEKDAY´LAST)
2) WEEKDAY´SUCC(WEEKDAY´LAST)
3) STONE´POS(QUARTZ)
b) Declare los siguientes tipos
1) frutas típicas.
2) proveedores de computadores.
c) Si el primer día del mes está en la variable D de tipo DAY, escriba una asignación que reemplace D por el día de la semana del N-ésimo día del mes.

4.7. Tipo Booleano

El tipo booleano es un tipo de enumeración predefinido, cuya declaración puede considerarse como

type BOOLEAN **is** (FALSE, TRUE);

Los valores booleanos son producidos por los operadores =, /=, <, <=, > y >=, los que tienen el significado normalmente aceptado y se aplican a muchos tipos. Por ejemplo:

```
if TODAY = SUN then
        TOMORROW := MON;
else
        TOMORROW:= DAY´SUCC(TODAY);
end if;
```

Existen otros operadores asociados al tipo BOOLEAN.

1. **not** Es un operador unario y cambia TRUE a FALSE y viceversa.
2. **and** Es un operador binario. El resultado es TRUE si ambos operadores son TRUE, en otro caso será FALSE.
3. **or** Es un operador binario. El resultado es TRUE si al menos uno de los operadores es TRUE, es FALSE si ambos operadores son FALSE.
4. **xor** Es un operador binario. El resultado es TRUE si y sólo si ambos operadores son distintos, es decir, uno es TRUE y el otro es FALSE.

Los operadores **and**, **or** y **xor** tienen la misma precedencia, la cual es menor que cualquier otro operador. En particular, tienen menor precedencia que =, /=, <, <=, > y >=. Como consecuencia de esto no se necesitan paréntesis para expresiones como

P < Q **and** I = J

Sin embargo, a pesar de tener la misma precedencia, los operadores **and**, **or** y **xor** no pueden mezclarse sin paréntesis, entonces

B **and** C **or** D

es incorrecto y debe escribirse

B **and** (C **or** D) o (B **and** C) **or** D

para enfatizar el sentido de la expresión.

Los programadores familiarizados con otros lenguajes recordarán que los operadores **and** y **or** tienen distinta precedencia. Esto puede conducir a errores, por ello Ada les da la misma precedencia y obliga a usar paréntesis. Obviamente, si se aplica el mismo operador en forma sucesiva no es necesario usar paréntesis ya que tanto **and** como **or** son asociativos. Por ejemplo:

B **and** C **and** D

es una expresión correctamente escrita.

Las variables y constantes booleanas pueden ser declaradas y utilizadas en la manera usual. Por ejemplo:

DANGER: BOOLEAN;

```
SIGNAL: COLOUR;
  . . .
DANGER := SIGNAL = RED;
```

La variable DANGER será TRUE si la señal es RED. Además, podríamos escribir

```
if DANGER then
        STOP_TRAIN;
end if;
```

Notemos que no deberíamos escribir

```
if DANGER = TRUE then
```

porque a pesar que es sintácticamente correcto no toma en cuenta que DANGER es una variable booleana y puede utilizarse directamente como una condición. Peor aún sería escribir

```
if  SIGNAL = RED then
        DANGER .= TRUE;
else
        DANGER := FALSE;
end if;
```

en lugar de `DANGER := SIGNAL = RED;`

Ejercicio:

1. Escriba declaraciones de constantes T y F con los valores TRUE y FALSE.
2. Evalúe (A /= B) = (A **xor** B) para todos los valores posibles de las variables booleanas A y B.

5. Estructuras de Control

Ada tiene tres estructuras de control: **if**, **case** y **loop**. A pesar que estas tres estructuras son suficientes para escribir programas con claridad, el lenguaje también incluye la instrucción **goto**.

La sintaxis de estas estructuras de control es similar. Existe una palabra reservada de inicio: **if**, **case** o **loop**, la que es pareada al final de la estructura por la misma palabra reservada precedida por la palabra **end**. Entonces tenemos

if	**case**	**loop**
…	…	…
end if;	**end case**,	**end loop**;

La palabra **loop** puede ir precedida por un cláusula de iteración que comienza con **for** o **while**.

5.1. Instrucción if

La sintaxis de esta instrucción es:

```
if_statement ::= if condition  then
                        sequence_of_statements
                 {elseif condition then
                        sequence_of_statements}
                 [else
                        sequence _of_statements]
                 end if;

condition::= boolean_expression
```

La semántica de la instrucción **if** de Ada es semejante a sus análogas en otros lenguajes. Veamos algunos ejemplos:

```
if HUNGRY then
        COOK;
        EAT;
        WASH_UP;
end if;
```

Esta es la variante más simple de la instrucción **if**. En este ejemplo la expresión booleana (condition) está dada por una variable booleana (HUNGRY) y las instrucciones (sequence_of_statements) son llamadas a procedimientos.

Las instrucciones internas al **if** (tanto en la rama **then** como en la **else**) pueden tener cualquier complejidad, incluso pueden ser instrucciones **if** anidadas. Por ejemplo, si deseamos encontrar las raíces de

$$ax^2 + bx + c = 0$$

Lo primero que hay que chequear es el valor de a. Si $a = 0$ la ecuación cuadrática se transforma en una ecuación lineal con una única raíz igual a $-c/b$. Si a no es cero se debe analizar el discriminante b^2-4ac para ver si las raíces son reales o complejas. Esto podría programarse de la siguiente manera

```
if A = 0.0 then
        -- caso lineal
else
        if B**2 - 4.0*A*C  >= 0.0 then
              -- raíces reales
        else
             -- raíces complejas
        end if;
end if;
```

Obsérvese la repetición de **end if**. Esto no es muy estético y ocurre con la suficiente frecuencia para ameritar una construcción especial: **elseif**, cuyo uso se muestra a continuación

```
if A = 0.0 then
        -- caso lineal
elseif B**2 - 4.0*A*C >= 0.0 then
        -- raíces reales
else
        -- raíces complejas
end if;
```

De esta forma se enfatiza el mismo status de los tres casos y la naturaleza secuencial de los chequeos. La rama **elseif** puede repetirse un número arbitrario de veces y la rama final **else** es opcional.

Ejercicio:

a) Las variables DAY, MONTH y YEAR contienen el día en curso, y están declaradas de la siguiente manera.

```
DAY: INTEGER range 1 .. 31;
MONTH: MONTH_NAME;
YEAR: INTEGER range 1901 .. 2099;
```

donde

```
type MONTH_NAME IS (JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,
                    NOV,DEC);
```

Escriba una sección de programa Ada que determine la fecha del día siguiente ¿Qué ocurrirá si la fecha es 31 DEC 2099?

b) X e Y son dos variables reales. Escriba un bloque que intercambie sus valores, si es necesario, para asegurar que el valor mayor quede en X.

5.2. Instrucción Case

Una instrucción **case** permite elegir una secuencia de instrucciones de entre varias alternativas de acuerdo al valor de una expresión. Por ejemplo, para controlar los movimientos de un vehículo robot podríamos utilizar

```
case ORDER is
        when LEFT => TURN_LEFT;
        when RIGHT => TURN_RIGHT;
        when BACK => TURN_BACK;
        when ON => null;
end case;
```

Es necesario explicitar todos los valores posibles (sólo una vez) para asegurar que no haya omisiones accidentales. Si para uno o más valores no se requiere una acción específica (como en el ejemplo) se debe usar la instrucción **null**, la cual no hace absolutamente nada, pero indica que eso es precisamente lo que se desea.

Es frecuente que se necesite realizar una misma acción para varios valores. Por ejemplo, para expresar la idea que desde lunes a jueves se trabaja; el viernes se trabaja y se asiste a una fiesta; y los sábados y domingos no hay actividades fijas, podemos escribir

```
case TODAY is
        when MON|TUE|WED|THU => WORK;
```

```
        when FRI                    => WORK;
                                       PARTY;
        when SAT|SUN                =>  null;
end case;
```

Si para varios valores sucesivos se realiza una misma acción, se pueden usar rangos, en el ejemplo anterior podría haberse escrito

```
when MON .. THU => WORK;
```

A veces se requiere que una cierta acción se realice para todos los valores que no hayan sido explícitamente indicados; esto se realiza utilizando la palabra reservada **others**, la cual puede aparecer sólo en la última alternativa. Según esto, el ejemplo anterior podría reescribirse

```
case TODAY is
        when MON|TUES|WED|THU => WORK;
        when FRI                    => WORK;
                                       PARTY;
        when  others                =>  null;
end case;
```

Ejercicio: Reescriba la respuesta al ejercicio (a) de la pág. anterior usando la instrucción case.

5.3. Instrucciones de iteración

La forma más simple de una instrucción de iteración es

```
loop
   sequence_of_statements
end loop;
```

Las instrucciones internas a la iteración se repiten indefinidamente hasta que una de ellas termina el loop de alguna manera. Por ejemplo, consideremos el cálculo del número trascendental *e*.

e = 1 + 1/1! + ½! + 1/3! + ¼! + . . .

Una posible solución sería

```
declare
        E:REAL := 1.0;
        I:INTEGER := 0;
        TERM:REAL :=1.0;
begin
        loop
          I:=I+1;
          TERM:= TERM / REAL(I);
           E:=E + TERM;
        end loop;
        . . .
end;
```

Notemos que la variable I es de tipo entero porque la lógica del problema nos indica que es un contador, pero el valor de *e* a calcular debe ser real, por eso es necesario hacer el cambio de entero a real al calcular el nuevo valor de TERM en cada iteración.

Matemáticamente hablando la serie arriba indicada tiende al valor de *e* cuando *n* tiende a infinito. Sin embargo, al realizar el cálculo computacional no es posible desarrollar la serie hasta el infinito, puesto que así nunca obtendríamos el valor requerido y, por otra parte, sabemos que la representación de los reales en un computador no es exacta, por lo que tampoco tiene sentido desarrollar la serie hasta un valor muy grande de *n*. La forma más sencilla de finalizar el **loop** es determinar una cantidad fija iteraciones antes del inicio del **loop**. Por ejemplo, supongamos que se decide realizar el **loop** N veces (N puede ser una constante o una variable a la que de alguna manera se le asigna un valor antes del loop), éste quedaría

```
loop
  if I = N then exit; end if;
  I:=I+1;
  TERM:= TERM / REAL(I);
   E:=E + TERM;
end loop;
```

La instrucción **exit** detiene el proceso de iteración, pasando el control a la instrucción inmediatamente siguiente a **end loop**. Puesto que la construcción

if condition **then exit**; **end if**;

es bastante usada, el lenguaje provee una construcción análoga más abreviada

exit when condition;

Por lo que el loop quedaría

```
loop
  exit when I = N;
  I:=I+1;
  TERM:= TERM / REAL(I);
   E:=E + TERM;
end loop;
```

La instrucción **exit** puede aparecer en cualquier parte del loop: al principio, al medio o al final. Además, puede aparecer más de una vez (en distintas ramas de instrucciones if, posiblemente, anidadas), sin embargo se recomienda estructurar el loop de forma tal que exista una sola sentencia **exit** dentro de cada loop. Es bastante común que la instrucción vaya al comienzo del loop (antes de cualquier otra instrucción) como en nuestro ejemplo. Para estos casos se puede usar la palabra reservada **while**, de la siguiente manera

```
while I /= N loop
  I:=I+1;
  TERM:= TERM / REAL(I);
   E:=E + TERM;
end loop;
```

En los casos en que se requiere iterar un número específico de veces se puede usar una última variante de la instrucción **loop**, la que consiste en la utilización de la palabra reservada **for**. Nuestro ejemplo quedaría

```
for  I in 1 .. N loop
   TERM:= TERM / REAL(I);
   E:=E + TERM;
end loop;
```

La variable usada para controlar la iteración (en este caso I) se declara implícitamente y no requiere ser definida externamente. Su tipo se determina por el tipo indicado en el rango de variación, y para los efectos internos del loop debe considerarse como una constante, en el sentido en que en cada iteración tiene un valor (determinado por el mecanismo del loop) que no puede ser modificado. Cuando se termina la iteración (por el término del rango o por una instrucción **exit**) I deja de existir.

Para tomar los valores del rango en orden descendente se utiliza la palabra reservada **reverse**, de la manera siguiente

for I **in reverse** 1 .. N **loop**

Nótese que el rango siempre se escribe en forma ascendente. No es posible definir un paso distinto de 1. Esto no debería causar problema porque la gran mayoría de los ciclos controlados por una variable lo hacen con un paso de 1 (ascendente o descendente) y aquellos que tienen un paso diferente (que son los menos) pueden ser simulados agregando los factores correspondientes. Por ejemplo, para calcular

$$\sum_{i=1}^{n} 1/2i$$

el código Ada más adecuado sería,

```
for  I in 1 .. N loop
   TERM:= 1.0 / (2.0 * REAL(I));
   E:=E + TERM;
end loop;
```

ya que refleja mejor la notación matemática estandar.

El rango en un **loop** tipo **for** puede ser vacío (por ejemplo si N es cero en el caso anterior) en cuyo caso el loop termina en forma inmediata. Los límites de un rango no necesariamente son constantes y/ o variables, son en general expresiones de un cierto tipo discreto (obviamente, el mismo para ambos límites). Las expresiones que determinan los límites son evaluadas una sola vez antes del inicio de la primera iteración y no pueden ser cambiados dentro del loop. Por este motivo el loop

```
N:=4;
for I in 1 .. N loop
     . . .
   N:=10;
end loop;
```

será ejecutado 4 veces a pesar que el valor de N haya sido cambiado a 10.

Como ya se dijo, el rango debe ser de algún tipo discreto, es decir, no necesariamente de tipo entero. Por ejemplo, para realizar una cierta acción para cada uno de los días de la semana escribiríamos

```
for TODAY in DAY loop
     . . .
end loop;
```

si quisieramos sólo realizar la acción para los días lunes a viernes escribiríamos

```
for TODAY in MON … FRI  loop
      . . .
end loop;
```

pero una manera más adecuada sería

```
for TODAY in WEEKDAY  loop
      . . .
end loop;
```

Como ya dijimos, la instrucción exit pasa el control al punto inmediatamente siguiente al loop dentro del cual se encuentra. Pero los loops pueden estar anidados y a veces ocurre que se desea salir de toda la estructura anidada y no solo del loop más interno. Por ejemplo, supongamos que estamos realizando una búsqueda en dos dimensiones.

```
for I in 1 .. N loop
        for J in 1 .. M loop
            -- si los valores de I y J satisfacen una cierta
            -- condición terminar la búsqueda
        end loop;
end loop;
```

Una instrucción exit simple nos sacaría del loop interno, pero estaríamos obligados a chequear nuevamente la condición (agreguese el inconveniente de que J ya no existe) inmediatamente después de terminado el loop interno. Este problema puede ser resuelto dando un nombre al loop externo y usando dicho nombre para salir inmediatamente de ambos.

```
SEARCH:
        for I in 1 .. N loop
                for J in 1 .. M loop
                        if condition_O_K then
                            I_VALUE :=  I;
                            J_VALUE := J;
                            exit SEARCH;
                         end if;
                         . . .
                end loop;
        end loop SEARCH;
--- el control pasa a este punto
```

Un loop es bautizado colocando un identificador antes de su inicio seguido de dos puntos. (Esto se parece mucho a la forma en que en otros lenguajes se definen labels, pero en Ada no es así, ya que no es posible usar los nombres de loops para la instrucción **Goto**.) El identificador debe ser repetido entre el correspondiente **end loop** y el punto y coma final.

La instrucción **exit** en su forma condicional también puede referenciar a un loop por su nombre.

```
exit SEARCH when condition;
```

Ejercicio:

a) Calcule

$$g = \sum_{p=1}^{n} \frac{1}{p} - \ln n$$

$$n \to \infty, g \to \gamma = 0.577215665$$

La función Ada LN entrega el logaritmo natural, el parámetro debe ser real y el resultado también es real.

b) Calcule *e* con un error absoluto de 0.00005. Es decir, la diferencia en valor absoluto entre dos valores (calculados) de *e* consecutivos debe ser menor o igual a 0.00005.

5.4. Instrucción goto y labels

A muchos podría sorprender el que un lenguaje de programación moderno contemple la instrucción **goto**, puesto que su uso se ha considerado una mala práctica de programación. Ada incluye esta instrucción, pero no estimula su uso al entregar estructuras de control suficientemente variadas para expresar cualquier flujo de control normalmente usado.

La generación automática de programas es el motivo principal para incluir la instrucción goto en Ada. Cuando se genera automáticamente un programa Ada desde una cierta especificación de más alto nivel, ésta última será considerada como "el programa fuente", por lo que el código Ada no tiene porque ser muy legible y es posible realizar cierta licencias, entre ellas el uso del goto. Además, al traducir un programa desde Cobol o Fortran a Ada puede resultar conveniente usar goto para hacer una traducción literal del programa original.

La instrucción goto en Ada tiene una sintaxis semejante a la de otros lenguajes, la palabra reservada **goto** seguido de un nombre de **label**, el cual se define entre paréntesis angulares dobles. Por ejemplo el label de nombre INICIO se define

```
<<INICIO>>
```

colocándolo en el lugar donde se desea que continúe el control del flujo de ejecución del programa. La instrucción en sí tendrá la siguiente forma

```
goto  INICIO;
```

Una instrucción **goto** no puede ser usada para transferir el control al interior de un **if**, **case** o **loop**, ni entre las ramas de las instrucciones **if** o **case**.

6. Tipos Compuestos

En esta sección se describirán los tipos compuestos, es decir, arreglos y registros. Además, con la introducción de los caracteres y strings se completará la discusión sobre los tipos de enumeración.

6.1. Arreglos

Un arreglo es un objeto compuesto que consiste de un grupo de componentes todos del mismo tipo. Un arreglo puede ser de una, dos o más dimensiones. Una declaración típica de un arreglo es

```
A: array (INTEGER range 1..6) of REAL
```

Aquí se declara la variable A como un objeto que contiene 6 componentes, cada uno de los cuales es de tipo REAL. Para referirse a los componentes individuales se debe indicar el nombre del array seguido de una expresión entre paréntesis que entrega un valor entero dentro del rango discreto 1..6. Si el resultado de esta expresión, conocida como el valor del índice, no está dentro del rango especificado, el sistema emitirá la excepción CONSTRAIN_ERROR. Para colocar ceros en cada componente de A podríamos escribir

```
for I in 1..6 loop
  A(I) := 0.0;
end loop;
```

Como se dijo, un arreglo puede tener varias dimensiones, en cuyo caso se debe indicar un rango para cada dimensión. Entonces

```
AA: array (INTEGER range 0..2, INTEGER range 0..3) of REAL;
```

es un arreglo de 12 componentes, cada uno de los cuales es referenciado mediante dos índices enteros, el primero dentro del rango 0..2 y el segundo en el rango 0..3. Para colocar todos estos elementos en cero podríamos escribir

```
for I in 0 .. 2 loop
   for J in 0 .. 3 loop
      AA(I,J) := 0.0;
   end loop;
end loop;
```

Los rangos no deben ser necesariamente estáticos, por ejemplo, es posible escribir

```
N: INTEGER:= . . .;
  . . .
B: array (INTEGER range 1..N) of BOOLEAN;
```

así, la cantidad de componentes de B estará determinada por el valor N, el que lógicamente deberá tener un valor antes de realizar la elaboración de B.

Los rangos discretos en los arreglos siguen reglas similares a los rangos de los sentencias **for**. Una de ellas es que un rango de la forma 1..6 implica que el tipo asociado es INTEGER, de ahí que podamos escribir

```
A: array (1..6) of REAL;
```

Sin embargo, el índice de un arreglo puede ser de cualquier tipo discreto. Podríamos tener por ejemplo

```
HOURS_WORKED: array (DAY) of REAL;
```

Este arreglo tiene siete componentes, desde HOURS_WORKED(MON) hasta HOURS_WORKED(SUN). Podemos usar este arreglo para almacenar las horas de trabajo para cada día de la semana

```
for D in WEEKDAY loop
  HOURS_WORKED(D) := 8.0;
end loop;
HOURS_WORKED(SAT) := 0.0;
HOURS_WORKED(SUN) := 0.0;
```

Si quisiésemos que el arreglo sólo tuviera cinco elementos, uno para cada día de lunes a viernes, podríamos escribir

```
HOURS_WORKED: array (DAY range MON..FRI) of REAL;
```

o mejor

```
HOURS_WORKED: array (WEEKDAY) of REAL;
```

Los arreglos tienen varios atributos relacionados con sus índices. A'FIRST y A'LAST entregan, respectivamente, los límites inferior y superior del primer (o único) índice de A. Entonces, de acuerdo a la última declaración de HOURS_WORKED

```
HOURS_WORKED'FIRST = MON
HOURS_WORKED'LAST = FRI
```

A'LENGTH entrega la cantidad de valores del primer (o único) índice de A.

```
HOURS_WORKED'LENGTH = 5
```

A'RANGE es una forma abreviada para A'FIRST .. A'LAST. Entonces

```
HOURS_WORKED'RANGE = MON .. FRI
```

Los mismos atributos pueden aplicarse a los demás índices de un arreglo multidimensional, para ello se debe agregar la dimensión (una expresión entera estática) involucrada entre paréntesis. Entonces, en el caso de nuestro arreglo bidimensional AA tenemos

```
AA'FIRST(1) = 0
AA'FIRST(2) = 0
AA'LAST(1) = 2
AA'LAST(2) = 3
AA'LENGHT(1) = 3
AA'LENGHT(2) = 4
AA'RANGE(1) = 0 .. 2
AA'RANGE(2) = 0 .. 3
```

El atributo RANGE es particularmente útil con iteraciones. Nuestros ejemplos anteriores quedarían mejor escritos como

```
for I in A'RANGE loop
  A(I) := 0.0;
end loop;


for I in AA'RANGE(1) loop
   for J in AA'RANGE(2) loop
     AA(I,J) := 0.0;
   end loop;
end loop;
```

El atributo RANGE también puede utilizarse en declaraciones, por ejemplo

```
J: INTEGER range A'RANGE;
```

es equivalente a

```
J: INTEGER range 1 .. 6;
```

En lo posible, es mejor utilizar los atributos para reflejar las relaciones entre entidades de un programa, de este modo los cambios quedan restringidos a las declaraciones sin que, en general, impliquen

cambios en el resto del código. Por ejemplo, si cambiamos los rangos de los arreglos A y AA, no será necesario modificar los ciclos **for** usados para asignarles valores.

Los arreglos que hemos visto hasta el momento son variables en el sentido normal. Por lo tanto, se les puede asignar valores y ser usadas en expresiones. Al igual que otras variables, a los arreglos se les puede asignar un valor inicial. Esto generalmente se denota mediante un conjunto que es la notación literal de un arreglo. La forma más simple de realizar esta asignación es mediante una lista ordenada (encerrada entre paréntesis) de expresiones (separadas por comas) que entregan los valores de los componentes. Por ejemplo, para inicializar el arreglo A con ceros, escribimos

```
A: array (1 .. 6) of REAL := (0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
```

En el caso de arreglos multidimensionales la lista será una lista con sublistas, por ejemplo

```
AA: array (0 .. 2, 0 .. 3) of REAL := ( (0.0,  1.0,  2.0,    3.0),
                                        (4.0,  5.0,  6.0,    7.0),
                                         (8.0,  9.0,  10.0, 11.0) );
```

De acuerdo a esta declaración, tenemos

```
AA(0,0) = 0.0
AA(0,1) = 1.0
AA(0,2) = 2.0
AA(0,3) = 3.0
AA(1,0) = 4.0
etc.
```

Nótese que la lista debe estar completa, es decir, si se desea inicializar un componente del arreglo, entonces deben inicializarse todos, y en el orden correcto.

Es posible declarar un arreglo como constante, en cuyo caso, obviamente, es obligatorio dar un valor inicial. Este tipo de arreglos es útil para implementar tablas. En el siguiente ejemplo se usa un arreglo constante para determinar si un día en particular es día de trabajo o no.

```
WORK_DAY: constant array (DAY) of BOOLEAN := (TRUE, TRUE, TRUE, TRUE, TRUE,
                                               FALSE, FALSE);
```

Un ejemplo interesante lo constituye un arreglo constante que permite determinar el día siguiente, sin preocuparse del último día de la semana.

```
TOMORROW: constant array (DAY) of DAY := (TUE, WED, THU, FRI, SAT, SUN, MON);
```

Finalmente, notemos que los componentes de un arreglo pueden ser de cualquier tipo o subtipo. Además, las dimensiones de un arreglo multidimensional pueden ser de diferentes tipos discretos. Un ejemplo un tanto extraño, pero válido, sería

```
STRANGE: array (COLOUR, 2 .. 7, WEEKDAY range TUE .. THU)
                        of PLANET range MARS .. SATURN;
```

Ejercicio:

1. Declare un arreglo F de enteros cuyo índice varíe entre 0 y N. Además, escriba código Ada para asignar a los componentes de F su correspondiente valor de la serie de Fibonaci.

$F_0 = 0,\ F_1 = 1, F_i = F_{i-1} + F_{i-2}$

2. Escriba código Ada para encontrar los índices I y J de componente con mayor valor del arreglo
A:**array** (1 .. N, 1 .. M) **of** REAL;

3. Declare un arreglo DAYS_IN_MONTH que permita determinar el número de días de cada mes.

4. Declare un arreglo YESTERDAY análogo al arreglo TOMORROW dado en un ejemplo anterior.

5. Declare un arreglo BOR tal que: BOR(P,Q) = P **or** Q

6. Declare una matriz constante unitaria UNIT de orden 3. Una matriz unitaria es aquella que tiene unos en su diagonal principal y ceros en todos los demás componentes.

6.2. Tipos arreglo

Los arreglos hasta ahora vistos no tienen un tipo explícito. De hecho son de tipo "anónimo". Reconsiderando el primer ejemplo del punto anterior, podemos escribir la definición de tipo

type VECTOR_6 **is array** (1 .. 6) of REAL;

y luego declarar A usando el tipo definido de la manera ya conocida

A: VECTOR_6;

Una ventaja de usar tipos es que es posible realizar asignaciones de arreglos completos que han sido declarados separadamente. Por ejemplo, si además declaramos

B:VECTOR_6;

podemos escribir

B:= A;

que sería una forma abreviada y más recomendable de

B(1):= A(1); B(2):= A(2); . . .B(6):= A(6);

Por otro lado, si escribimos

C: **array** (1 .. 6) **of** REAL;
D: **array** (1 .. 6) **of** REAL;

la expresión C:= D; será ilegal debido a que C y D no son del mismo tipo. Son de diferentes tipos, ambos anónimos. La regla que subyace en todo esto es que cada definición de tipo introduce un nuevo tipo y en este caso la sintaxis nos dice que una definición de tipo arreglo es el trozo de texto que comienza con la palabra **array** hasta el punto y coma (exclusive). Incluso, si escribimos

C,D: **array** (1 .. 6) **of** REAL;

la expresión C:= D; seguirá siendo incorrecta, esto porque esta declaración múltiple es sólo una forma abreviada de las dos declaraciones anteriores.

El que se defina o no un nuevo tipo depende mucho del grado de abstracción que se pretenda lograr. Si se está pensando en los arreglos como objetos que serán manipulados (asignados, mezclados, etc..) entre ellos, se hace necesario definir un tipo. Si, por otro lado, se conceptualiza al arreglo sólo como una agrupación de elementos indexables, sin relación alguna con otros arreglos semejantes, entonces lo más recomendable es definirlo como de tipo anónimo.

El modelo de tipos arreglo presentado no es del todo satisfactorio, pues no nos permite representar una abstracción que agrupe arreglos con diferentes límites, pero que para todos los demás efectos forman una sola familia. En particular, no nos permite escribir subprogramas que puedan tomar arreglos de largo variable como parámetros reales. De ahí que Ada introduce el concepto de tipo arreglo no restringido, en el cual no se expresan los límites al momento de la declaración. Consideremos

type VECTOR **is array** (INTEGER **range** <>) **of** REAL;

(Al símbolo compuesto <> se le denomina "caja").

Con esto se está diciendo que VECTOR es el nombre de un tipo arreglo unidimensional de componentes REAL con índice INTEGER. Sin embargo, los límites del rango no se han especificado. Esta información debe (obligatoriamente) aportarse cuando se declaran variables o constantes de tipo VECTOR. Esto puede hacerse de dos formas. Podemos introducir un subtipo intermedio y luego declarar los objetos (variables y/o constantes).

subtype VECTOR_5 **is** VECTOR(1..5);
V: VECTOR_5;

o podemos declarar los objetos directamente

```
V:VECTOR(1 .. 5);
```

En ambos casos los límites deben darse mediante una restricción de índices que toman la forma de un rango discreto entre paréntesis.

El índice del tipo arreglo no restringido también puede darse a través de un subtipo. Entonces, si tenemos

```
type P is array (POSITIVE range <>) of REAL;
```

los límites reales de cualquier objeto deben estar dentro del rango estipulado por el subtipo POSITIVE.

Otra definición de tipo muy útil es

```
type MATRIX is array (INTEGER range <>, INTEGER range <>) of REAL;
```

la que luego nos permite definir subtipos como

```
subtype MATRIX_3 is MATRIX(1 ..3, 1 .. 3);
```

y también variables

```
M:MATRIX(1 .. 3, 1 .. 3);
```

Volvamos al asunto de la asignación entre arreglos completos. Para realizar este tipo de asignación es necesario que los arreglos ubicados a ambos lados del comando de asignación sean del mismo tipo y que sus componentes puedan ser pareados. Esto no significa que los límites tengan que ser necesariamente iguales, sino sólo que la cantidad de componentes en las dimensiones correspondientes sea la misma. De ahí que podamos escribir

```
V:VECTOR(1 .. 5);
W:VECTOR(0 .. 4);
  . . .
V:= W;
```

Tanto V como W son de tipo VECTOR y tienen 5 componentes. También sería válido escribir

```
P:MATRIX(0 .. 1, 0 .. 1);
Q: MATRIX(6 .. 7, N .. N+1);
   . . .
P:= Q;
```

La igualdad y diferencia entre arreglos siguen reglas similares a la asignación. Dos arreglos pueden ser comparados sólo sin son del mismo tipo, y son iguales si sus dimensiones correspondientes tienen el mismo número de componentes y si los componentes pareados son iguales.

Los atributos FIRST, LAST, LENGHT y RANGE también pueden aplicarse a tipos y subtipos en la medida estén restringidos, entonces

```
VECTOR_6'LENGHT = 6
```

pero

```
VECTOR'LENGHT   es ilegal.
```

6.3. Caracteres y strings

Completaremos la discusión de los tipos de enumeración introduciendo los tipo caracter. En los tipos de enumeración vistos anteriormente, tal como

type COLOUR **is** (RED, AMBER, GREEN);

los valores habían sido representados por identificadores. También es posible tener un tipo de enumeración en el que parte o todos los valores están representados por literales de caracter.

Un literal de caracter es otra forma que puede tomar un lexema. Consiste de un caracter único encerrado entre apóstrofes. El caracter debe ser un caracter imprimible o un espacio blanco. No puede ser un caracter de control, tales como tabulador horizontal o nueva línea.

Este es un caso donde se hace distinción entre mayúsculas y minúsculas. De ahí que los literales

'A' y 'a'

sean diferentes.

Basados en lo dicho, podemos definir el tipo de enumeración.

type ROMAN_DIGIT **is** ('I', 'V', 'X', 'L', 'C', 'D', 'M');

y luego declarar

DIG: ROMAN_DIGIT:= 'D';

Además, son aplicables todos los atributos de los tipos de enumeración.

ROMAN_DIGIT'FIRST = 'Y'
ROMAN_DIGIT'SUCC('X') = 'L'
ROMAN_DIGIT'POS('M') = 6

Existe un tipo de enumeración predefinido de nombre CHARACTER que es (obviamente) el tipo caracter. Su declaración es aproximadamente

type CHARACTER **is** (*null*, . . ., 'A', 'B', 'C', . . ., *del*);

Nótese que la introducción de los tipos ROMAN_DIGIT y CHARACTER lleva a la sobrecarga de algunos literales. De ahí que la expresión

'X' < 'L'

sea ambigua. No sabemos si se están comparando caracteres de tipo ROMAN_DIGIT o CHARACTER. Para resolver esta ambigüedad se debe cualificar uno o ambos argumentos

CHARACTER'('X') < 'L' = FALSE
ROMAN_DIGIT'('X') < 'L' = TRUE

Así como existe el tipo predefinido CHARACTER, existe el tipo predefinido STRING

type STRING **is array** (POSITIVE **range** <>) **of** CHARACTER;

Este es un tipo arreglo normal y por lo tanto sigue las reglas previamente enunciadas. Por lo tanto, podemos escribir

S:STRING(1 .. 7);

para declarar un arreglo de rango 1 .. 7. En el caso de arreglos constantes es posible deducir los límites a partir del valor inicial. Entonces, si escribimos

G: **constant** STRING:= ('P', 'I', 'G');

el límite inferior de G (es decir, G'FIRST) será 1, puesto que el tipo del índice de STRING es POSITIVE y POSITIVE'FIRST es 1.

Existe una notación alternativa más abreviada para los strings, esta consiste en una cadena de caracteres encerrada entre comilla dobles. El ejemplo anterior quedaría

G: **constant** STRING:= “PIG”;

El uso más importante de los strings es, obviamente, la creación de textos de salida. Es posible “imprimir” una secuencia simple de caracteres usando el subprograma (sobrecargado) PUT. Por ejemplo, la llamada

PUT(“The Countess of Lovelace”);

colocará el texto

The Countess of Lovelace

en algún archivo apropiado.

Es posible aplicar los operadores relacionales <, <=, >, y >= sobre los string. Las reglas son las ampliamente usadas en distintos lenguajes. Por ejemplo, todas las comparaciones siguientes son verdaderas.

“CAT” < “DOG”
“CAT” < “CATERPILLAR”
“AZZ” < “B”
“” < “A”

Existe, además, el operador de concatenación &, que permite unir dos strings para generar uno nuevo. El límite inferior del resultado es igual al límite inferior del primer operando. Por ejemplo, si tenemos

STRING_1:STRING:=“CAT”;
STRING_2:STRING(11 .. 18):=“ERPILLAR”;

entonces

STRING_1 & STRING_2 = “CATERPILLAR”
STRING_2 & STRING_1 = “ERPILLARCAT”

es decir, dos strings de igual largo, pero con rangos diferentes, en la primera expresión el rango es 1 .. 11 y la segunda 11 .. 21. En todo caso, cuando se están generando strings para salidas los rangos en sí no son relevantes.

Finalmente, es posible tomar sólo una parte de un string, ya sea para utilizarlo en una expresión, o para asignar ciertos valores a una parte específica de un string. Por ejemplo, si escribimos

```
S:STRIG(1 .. 10):= “0123456789”;
  . . .
T: constant STRING:=S(3 .. 8);
  . . .
S(1 .. 3):= “ABC”;
```

al final T tendrá el valor “234567”, con T’FIRST = 3 y T’LAST = 8. Y S terminará con el contenido “ABC3456789”.

Ejercicio:

1. Declare un arreglo constante de nombre ROMAN_TO_INTEGER que pueda ser usado como una tabla para convertir un ROMAN_DIGIT en su entero equivalente. Por ejemplo, para convertir ‘C’ en 100.

2. Escriba código Ada que tome un objeto R de tipo ROMAN_INTEGER y lo calcule el valor entero correspondiente en la variable V. Debe asumirse que R es un número romano correctamente escrito.

6.4. Registros

Un registro es un objeto compuesto cuyos componentes tienen nombres y pueden ser de tipos distintos. A diferencia de los arreglos, no es posible tener registros de tipo anónimo. Es obligación definir un tipo registro explícito para posteriormente crear los objetos (constantes y/o variables) correspondientes. Consideremos el siguiente tipo registro

```
type MONTH_NAME is (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG,
                    SEP, OCT, NOV, DEC);

type DATE is
    record
      DAY:INTEGER range 1 .. 31;
      MONTH:MONTH_NAME;
      YEAR:INTEGER;
    end record;
```

DATE es un tipo registro que contiene tres componentes con nombre: DAY, MONTH y YEAR. Ahora, podemos definir variables y constantes de la manera usual.

```
D:DATE;
```

declara una variable de tipos DATE. Mediante la notación punto se pueden accesar los componentes individuales de D. Así, para asignar valores a estos componentes podemos escribir

```
D.DAY:= 15;
D.MONTH:=AUG;
D.YEAR:= 1959;
```

Los registros pueden ser manipulados como objetos completos. Además, es posible asignar valores a todos los componentes de un registro mediante un conjunto ordenado. Por ejemplo

```
D:DATE:= (15,AUG,1959);
E:DATE;
  . . .
E:= D;
```

Es posible dar valores por omisión a todos o algunos de los componentes. Por ejemplo, al escribir

```
type COMPLEX is
    record
      RL: REAL:= 0.0;
      IM: REAL:= 0.0;
    end record;
```

se está declarando un tipo que contiene dos componentes de tipo REAL y cada uno de ellos se le da un valor por omisión de 0.0. Ahora podemos definir

```
C1:COMPLEX;
C2: COMPLEX:= (1.0, 0.0);
```

La variable C1 tiene sus componentes con los valores por omisión, es decir, 0.0. Por su parte, la variable C2 recibe valores que reemplazan a los indicados en la definición del tipo. Nótese que a pesar que el segundo componente de C2 toma el mismo valor estipulado por omisión, es necesario indicarlo explícitamente. Esto se debe a que cuando se asignan los valores mediante un conjunto, es obligación dar el valor para todos y cada uno de los componentes.

Las únicas operaciones predefinidas para los registros son = y /=, y la asignación. Es posible realizar otras operaciones a nivel de registro o definirlas explícitamente en subprogramas, como se verá más adelante.

Los componentes de un registro pueden ser de cualquier tipo; entre otros pueden ser otros registros o arreglos. Sin embargo, si el componente es un arreglo, este debe ser restringido y no puede ser de tipo anónimo. Y, obviamente, un registro no puede contener una instancia de sí mismo.

Los componentes no pueden ser constantes, pero un registro en sí si puede serlo. Por ejemplo, para representar la unidad imaginaria (raíz cuadrada de -1) podemos declarar

```
I: constant COMPLEX:= (0.0, 1.0);
```

A continuación presentamos un ejemplo más elaborado de un registro

```
type PERSON is
        record
          BIRTH: DATE;
          NAME:STRING (1 .. 20);
        end record;
```

El registro PERSON tiene dos componentes, el primero es otros registro y el segundo es un arreglo. Ahora podemos escribir, por ejemplo

```
JOHN:PERSON;
JOHN.BIRTH:=(19, AUG, 1937);
JOHN.NAME(1 .. 4):="JOHN";
```

y tendremos

```
JOHN := ((19, AUG, 1937), "JOHN          ");
```

Ejercicio:

1. Declare tres variables C1, C2 y C3 de tipo COMPLEX; y escriba uno o más sentencias para realizar la suma y el producto de C1 y C2, guardando el resultado en C3.

7. Subprogramas

En Ada existen dos tipos de subprogramas: funciones y procedimientos. Las funciones retornan un valor y son usadas en expresiones, mientras que los procedimientos no retornan valor y son llamados como instrucciones.

Las acciones a realizar en un subprograma se describen dentro de lo que se denomina "cuerpo del subprograma", el que es declarado de la manera usual en la parte declarativa de, por ejemplo, un bloque u otro subprograma.

7.1. Funciones

Todas las funciones comienzan con la palabra reservada **function** seguida del nombre (identificador) de la función. Si existen parámetros, después del identificador se entrega una lista con los parámetros (separados por ";") encerrada entre paréntesis. Luego de la lista de parámetros (si existe) viene la palabra **return** y el tipo (o subtipo) del valor de retorno de la función. Tanto el tipo de los parámetros como del valor de retorno debe ser indicado con un identificador de tipo (o subtipo) declarado previamente. Por ejemplo, no es posible indicar que el valor de retorno será un cierto rango de los enteros indicándolo explícitamente luego de la palabra **return**, sino que es necesario definir un subtipo para dicho rango con anterioridad a la declaración de la función y escribir el nombre del subtipo a continuación de la palabra **return**.

A la parte descrita hasta ahora se le conoce como "especificación de la función" y es la que entrega los datos para el entorno, en el sentido que en ella se entrega la información necesaria y suficiente para llamar a (hacer uso de) la función.

Después de la especificación viene la palabra **is** seguida del cuerpo de la función, que es semejante a un bloque: una parte declarativa, **begin**, una secuencia de instrucciones y **end**. Como en el caso de los bloques, la parte declarativa puede no existir, pero al menos debe existir una instrucción.

En algunos casos (como veremos más adelante) es necesario escribir sólo la especificación de la función, sin su cuerpo. En este caso en lugar de la palabra **is** va un ";".

Los parámetros formales son objetos locales de una función y actúan como constantes cuyos valores iniciales son calculados de acuerdo a los correspondientes parámetros reales. Cuando una función es llamada (utilizada dentro de un expresión) se elabora la parte declarativa de la manera usual y luego se ejecutan las instrucciones. Para entregar el valor de retorno se utiliza la instrucción **return**, la cual además entrega el control al lugar desde donde se hizo la llamada.

Consideremos nuestro ejemplo de la raíz cuadrada:

```
function SQRT(X:REAL) return REAL is
        R:REAL;
begin
        --  calcular valor de la raíz de X y guardarlo en R
        return R;
end SQRT;
```

La función puede ser utilizada, por ejemplo, en

```
T:REAL:= 0.3;
  . . .
S:= SQRT(T + 0.5) + 3.6;
```

para ello se evalúa la expresión T + 0.5 (es decir, T + 0.5 es el parámetro real) y se asigna al parámetro formal X, el que dentro de la función se comporta como una constante, con valor inicial 0.8. Esto equivale a tener

```
X: constant REAL := T + 0.5;
```

Luego se elaboran las declaraciones, si las hubiera, y finalmente se ejecutan las instrucciones. La última de las cuales es la instrucción **return**, la que retorna el control a la expresión "SQRT(T + 0.5) + 3.6" junto con el valor contenido en R.

La expresión de una instrucción **return** puede ser de cualquier nivel de complejidad, lo importante es que el resultado de dicha expresión sea del tipo indicado como tipo de retorno en la especificación de la función.

El cuerpo de una función puede contener más de una instrucción **return**. La ejecución de cualquiera de ellas termina la función. Por ejemplo,

```
function SIGN(X:INTEGER) return INTEGER is
-- Entrega +1, 0 o -1 según sea el signo del entero X
begin
        if X > 0 then
                return +1;
        elseif X < 0 then
                return  -1;
        else
                return 0;
        end if;
end SIGN;
```

Puede verse que la última instrucción no es necesariamente un **return** , lo importante es que la semántica del cuerpo de la función considere una instrucción **return** para todos los posibles casos. Si no fuese así y se llegase "**end** SIGN;" el sistema entregaría la excepción PROGRAM_ERROR (este tipo de excepción se usa para situaciones en las cuales se ha violado la secuencia de control en tiempo de ejecución).

Notemos que cada llamada a una función genera una nueva instancia de cada uno de los objetos declarados en ella (incluyendo, lógicamente, los parámetros) y éstos desaparecen cuando la función termina. Por este motivo (es decir, la administración dinámica de memoria) es posible llamar recursivamente a una función. Por ejemplo:

```
function FACTORIAL(N:POSITIVE) return POSITIVE is
begin
        if N = 1 then return 1;
        else     return N * FACTORIAL(N-1);
        end if;
end FACTORIAL;
```

Si escribimos

```
F:= FACTORIAL(4);
```

se produce la llamada recursiva a la función FACTORIAL con parámetros reales 4, 3, 2 y 1. A veces se dice que una función es recursiva puesto que "se llama a si misma". Sin embargo, es necesario entender que cuando se llama a FACTORIAL(4) se genera una instancia de la función, es decir, se reserva espacio de memoria para todos los objetos locales (más otros datos) y se procede a ejecutar las instrucciones. Cuando se llega a "4*FACTORIAL(4 - 1)", se genera otra instancia de la función, es decir, se toma más espacio de memoria para localizar los objetos locales, por lo tanto en este punto hay dos áreas de memoria para almacenar el estado de cada una de las dos llamadas (FACTORIAL(4) y FACTORIAL(3)). Esto se repetirá, en este caso, cuatro veces. Al ejecutar el cuerpo de la llamada FACTORIAL(1) no se genera una nueva instancia, sino que retorna el valor positivo 1. En este momento, la llamada FACTORIAL(1) termina, sus objetos locales dejan de existir, con lo que el espacio para ella reservado también desaparece. Lo mismo ocurrirá para las llamadas FACTORIAL(2), FACTORIAL(3) y finalmente FACTORIAL(4). Lógicamente, no es necesario repetir el código de la función, cada instancia de la función controla el número de instrucción que se está ejecutando.

No es necesario chequear que el parámetro sea positivo, pues el parámetro formal N es del subtipo POSITIVE. Por lo tanto, si se intenta FACTORILA(-2) se obtendrá CONSTRAIN_ERROR. Sin embargo, si intentamos FACTORIAL(1000) podría generarse una excepción STORAGE_ERROR puesto que las instancias de mil llamadas a la función estarían presentes en un cierto momento. Por otro lado, la llamada FACTORIAL(70) podría generar la excepción NUMERIC_ERROR..

Como ya se indicó un parámetro formal puede ser de cualquier tipo, pero dicho tipo (o subtipo) debe tener un nombre. Por lo tanto, está permitido que los parámetros puedan ser de tipo array no restringido. Por ejemplo, si definimos el tipo VECTOR como

```
type VECTOR is array (INTEGER range <>) of REAL;
```

podemos escribir la función

```
function SUM(A:VECTOR) return REAL is
        RESULT: REAL:= 0.0;
begin
        for I in A´RANGE loop
                RESULT := RESULT + A(I);
        end loop;
        return RESULT;
end SUM;
```

En este caso los límites del vector A serán tomados del arreglo de tipo VECTOR usado como parámetro real. Recordemos que todas las variables y constantes del tipo VECTOR deben tener límites definidos.

Entonces podemos escribir

```
V:VECTOR(1 .. 4):= (1.0,2.0,3.0,4.0);
W:VECTOR(-1 .. 3):= (1.5,2.5,3.5,4.5,5.5);
S:REAL:=SUM(V);
T:REAL;
 . . .
T:=SUM(W);
```

con lo que S tomará el valor 10 y T el valor 17.5.

Lógicamente, en Ada una función puede tener parámetros de tipo array restringido, pero debe ser por medio de un nombre de tipo o subtipo. Por ejemplo, sería incorrecto escribir

```
function SUM_6(A:VECTOR(1 .. 6)) return REAL
```

debería definirse un tipo o subtipo y luego usarlo para indicar la naturaleza del parámetro. Por ejemplo:

```
type VECTOR_6_A is array (1 ..  6) of REAL;
subtype VECTOR_6_B is VECTOR(1..6);
function SUM_6_A(A:VECTOR_6_A) return REAL;
function SUM_6_B(A:VECTOR_6_B) return REAL;
```

Lógicamente, el elegir entre definir un nuevo tipo o un subtipo dependerá de los requerimientos del problema a resolver. La diferencia entre SUM_6_A y SUM_6_B radica en que la segunda podrá aceptar arreglos de tipo VECTOR_6_B, VECTOR y otros subtipos de éste (en la media que contengan 6 elementos indexados del 1 al 6), en cambio SUM_6_A sólo aceptará arreglos del tipo VECTOR_6_A.

Consideremos otro ejemplo:

```
function INNER(A,B. VECTOR) return REAL is
        RESULT: REAL := 0.0;
begin
        for I in A´RANGE loop
                RESULT.= RESULT + A(I) * B(I);
        end loop;
        return RESULT;
end INNER;
```

La función INNER calcula el producto interno entre dos vectores A y B. Tenemos aquí un ejemplo de una función con más de un parámetro.

La función INNER no es un código robusto, puesto que sólo funciona correctamente cuando ambos parámetros A y B tiene los mismos límites y no se controla los casos en que éstos difieren. Por ejemplo:

```
T:VECTOR(1 .. 3):= (1.0, 2.0, 3.0);
U:VECTOR(1 .. 3):= (2.0, 3.0, 4.0);
V:VECTOR(0 .. 2):= (3.0, 4.0, 5.0);
W:VECTOR(1 .. 4):= (4.0, 5.0, 6.0, 7.0);
 . . .
R:=INNER(T,U);   -- R=1.0*2.0 + 2.0*3.0 + 3.0*4.0 = 20.0
R:=INNER(T,V);    -- CONSTRAIN_ERROR al intentar accesar  B(3)
R:=INNER(T,W);   -- R=1.0*4.0 + 2.0*5.0 + 3.0*6.0 = 32.0
```

En el tercer caso se obtiene un valor (32.0), pero es erróneo calcular el producto interno de dos vectores de distinto largo. Sería deseable que el lenguaje proveyera un mecanismo para chequear en el momento de la llamada que los límites de A y B coincidan, pero lamentablemente no es así. La solución más adecuada es verificar los límites al comienzo de la función y, posiblemente, generar explícitamente una excepción CONSTRAIN_ERROR.

```
if A´FIRST /= B´FIRST or A´LAST /= B´LAST then
        raise CONSTRAIN_ERROR;
end if;
```

Hemos visto que un parámetro formal puede ser un arreglo no restringido, pero el valor de retorno de una función también puede ser un arreglo cuyos límites se definen de acuerdo a los del arreglo entregado por

la instrucción **return**. Por ejemplo, la siguiente función retorna un arreglo con los mismos límites del parámetro, pero con los elementos en orden inverso.

```
function REV(X:VECTOR) return VECTOR is
        R:VECTOR(X´RANGE);
begin
        for I in X´RANGE loop
                R(I):= X(X´FIRST + X´LAST -I);
        end loop;
        return R;
end REV;
```

Ejercicio:

1. Escriba una función de nombre PAR que indique si un entero es par (TRUE) o impar (FALSE).
2. Reescriba la función FACTORIAL de forma tal que el parámetro pueda ser positivo o cero (use el subtipo NATURAL). Recuerde que FACTORIAL(0) = 1.
3. Escriba la función OUTER que entrega el producto externo de dos vectores (posiblemente con distintos rangos). El producto externo de los vectores A y B se define como la matriz $C_{ij} = A_i * B_j$.
4. Escriba la función MAKE_UNIT que toma un valor positivo N y entrega una matriz unitaria real de NxN. Recuerde que una matriz unitaria es aquella que contiene unos un su diagonal principal y ceros en todos los demás elementos.
5. Escriba la función MCD (usando recursión) que entrega el máximo común divisor de dos enteros no negativos. Use el algoritmo de Euclides.

   mcd(x,y) = mcd(y, x mod y) si $y \neq 0$

   mcd (x,0) = x

7.2. Operadores

Anteriormente dijimos que toda función comienza con la palabra reservada **function** seguida de un identificador, sin embargo también es posible usar como nombre de función un string de caracteres, siempre y cuando sea alguno de los siguientes operandos (entre comillas).

and	**or**	**xor**		
=	<	<=	>	>=
+	-	&	**abs**	**not**
/	*	**mod**	**rem**	**

En estos casos la función definirá un nuevo significado a los respectivos operadores. Para ejemplificar esto podemos reescribir la función INNER de la siguiente manera

```
function "*" (A,B:VECTOR) return REAL is
        RESULT: REAL := 0.0;
begin
        for I in A´RANGE loop
                RESULT:= RESULT + A(I) * B(I);
        end loop;
        return RESULT;
end "*";
```

Ahora podemos usar la nueva función con la sintaxis propia del operador *. Entonces, en lugar de

```
R:=INNER(V,W);
```

podemos escribir

```
R:= V * W;
```

(Nótese que de todas maneras se puede usar la notación infijada R:= "*"(V,W), pero no es recomendable).

Este nuevo significado del operador * se diferencia de los ya existentes (multiplicación de reales y enteros) por el contexto dado por los tipos de los parámetros V y W, y por el tipo requerido por R.

Al hecho que un cierto operador tenga varios significados se le conoce como sobrecarga (overloading). La sobrecarga de operadores predefinidos no es nuevo, ha existido en los lenguajes de alto

nivel desde hace mucho tiempo, lo nuevo de Ada es que permite que el programador también pueda realizar sobrecarga de operadores, funciones y procedimientos de acuerdo a las necesidades del problema a resolver.

A pesar que los operadores pueden ser sobrecargados, no está permitido el cambiar la sintaxis de la llamada, ni cambiar su nivel de precedencia (jerarquía) con respecto a los demás operadores. Por ejemplo, el operador * siempre debe tener dos parámetros, los cuales pueden ser de cualquier tipo y el resultado puede ser de cualquier tipo. Es decir, la sintaxis (forma de escribir) se mantiene, pero la semántica (acción del operador) puede cambiar.

Dos operadores (funciones, procedimientos) sobrecargados son DIFERENTES. Si no entendemos esto claramente podríamos considerar que la función "*" recién definida es recursiva. Pero puesto que el producto de vectores es diferente (es otro operador con el mismo nombre) al producto de reales, tenemos que A(I)*B(I) no es una llamada recursiva, sino una llamada al producto de reales dentro del producto de vectores. Existe un riesgo de escribir accidentalmente operadores recursivos cuando se está reemplazando un operador preexistente en lugar de crear uno nuevo.

Notemos que dos operadores sobrecargados existen simultáneamente, ninguno de ellos oculta al otro como ocurre cuando, por ejemplo, una variable es declarada con un mismo nombre dentro de un bloque (o loop) interno. En este último caso la variable más interna "cubre" a la interna. Dos (o más) operadores sobrecargados coexisten y se sabe cuando se está usando uno u otro de acuerdo al contexto. Por ejemplo:

```
declare
        U,V,W:VECTOR;
        A,B,C,E:REAL;
        I:INTEGER;
begin
        --- se asignan valores adecuados a V, W, B y C.
        A:=V*W;                 -- correcto
        A:=B*C;                 -- correcto
        B:=B*REAL(I);           -- correcto
        U:=V*W;                 -- incorrecto
        I:=B*C;                 -- incorrecto
        E:= 1.0 + A*C;          -- correcto
        A:= C*W;                -- incorrecto
end;
```

Por último notemos que para efecto de nombrar operadores las mayúsculas y minúsculas no se diferencian, por ejemplo, al sobrecargar el operador **or** se puede usar "or" o "OR" o incluso "Or".

Ejercicio:

1. Escriba las funciones "+" y "*" para sumar y multiplicar dos valores de tipo complejo.
2. Escriba las funciones "+" y "*" pasa sumar y multiplicar un complejo con un real. Suponga que el real siempre es el segundo operando.

7.3. Procedimientos

Los procedimientos son subprogramas que no retornan un valor y son llamados como instrucciones. La sintaxis es similar a la de las funciones. Se comienza con una palabra reservada (**procedure** en este caso), luego viene un identificador (que no puede ser un operador) , le sigue la lista de parámetros, no existe un tipo de retorno y el resto es equivalente a lo dicho para las funciones.

Los parámetros pueden ser de tres tipos: **in**, **out** o **in out**. Si no se indica el modo se asume que el parámetro es de entrada (**in**).

1) **in**: El parámetro formal es una constante y se permite sólo la lectura del valor asociado al parámetro real.
2) **in out**: El parámetro formal es una variable y se permite tanto la lectura como la actualización del valor asociado al parámetro real.
3) **out**: El parámetro formal es como una variable, se permite sólo la actualización del valor asociado al parámetro real. No se puede leer su valor.

En el caso de las funciones sólo puede haber parámetros de entrada, por lo que los ejemplos anteriores podrían escribirse

```
function SQRT(X: in REAL) return REAL;
function "*" (A,B: in VECTOR) return REAL;
```

Veamos el funcionamiento de los modos **in** y **out**.

```
procedure ADD(A,B: in INTEGER; C: out INTEGER) is
```

```
begin
        C:=A+B;
end;
```

con

```
P,Q:INTEGER;
 . . .
ADD(2+P,37,Q);
```

Al ser llamado el procedimiento ADD, en primer lugar, se evalúan las expresiones 2+P y 37 (en cualquier orden) y los respectivos resultados son pasados a A y B que en adelante se comportan como constantes. Luego se evalúa A+B y el valor se asigna al parámetro formal C. Al terminar el valor de C se asigna a la variable Q. Esto equivale, más o menos, a haber escrito.

```
declare
        A: constant INTEGER:= 2+P;      -- in
        B: constant  INTEGER := 37;     -- in
        C:INTEGER;                      -- out
begin
        C:=A+B;                         -- cuerpo
        Q:=C;                           -- out
end;
```

Veamos un ejemplo con modo **in out**.

```
procedure INCREMENT(X: in out INTEGER) is
begin
        X:=X+1;
end;
```

con

```
I:INTEGER;
 . . .
INCREMENT(I);
```

Al llamar a ADD el valor de I es asignado a la variable X, luego el valor de X se incrementa en 1. Al terminar, el nuevo valor de X se asigna al parámetro real I. Esto equivale a haber escrito

```
declare
        X:INTEGER.=I;
begin
        X:=X+1;
        I:=X;
end;
```

Para cualquier tipo escalar (como es el caso de los INTEGER) el modo **in** equivale a copiar un valor en la llamada, el modo **out** equivale a copiar un valor al terminar de ejecutar el procedimiento y el modo **in out** equivale a la composición de los anteriores.

Si el modo es **in**, entonces el parámetro real puede ser cualquier cualquiera expresión del tipo indicado en el parámetro formal. En los otros casos el parámetro real debe ser necesariamente una variable del tipo adecuado. La identidad de dicha variable queda determinada cuando se produce la llamada y no puede ser cambiada dentro del procedimiento. Por ejemplo:

```
I:INTEGER;
A:array (1 .. 10) of INTEGER;
procedure SILLY (X: in out INTEGER) is
begin
        I:= I+1;
        X:=X+1;
end;
```

entonces al ejecutar

```
A(5):=1;
I:=5;
SILLY(A(I));
```

el valor final en A(5) será 2, I tomará el valor 6, pero A(6) no será afectado. En otras palabras, al momento de la llamada la variable X se asocia a A(5) no a A(I), por lo que cualquier cambio en I no influye en X.

Todos los parámetros (**in**, **out** o **in out**) y los valores de retorno de las funciones deben pertenecer a un cierto tipo (o subtipo) y cumplir todas las restricciones que en la definición del tipo (o subtipo) correspondiente se hayan especificado.

Si un parámetro formal es de tipo array restringido (es decir, sus límites están predeterminados), los límites del parámetro formal deben coincidir, no basta con que el número de componentes sea idéntico en cada una de las dimensiones. Como se puede ver, el mecanismo de paso de parámetros es más riguroso que la instrucción de asignación. Para los arreglos no restringidos los límites se toman de los parámetros reales, incluso para el modo **out**.

Consideremos otro ejemplo, un procedimiento que resuelve la ecuación $ax^2 + bx + c = 0$

```
procedure QUADRATIC(A,B,C:in REAL; ROOT_1,ROOT_2: out REAL;
                    OK:out BOOLEAN) is
    D:constant REAL:=B**2 - 4*A*C;
begin
    if  D < 0.0 or  A = 0.0 then
        OK:=FALSE;
        return;
    end if;
    ROOT_1:=(-B+SQRT(D))/(2.0*A);
    ROOT_2:=(-B-SQRT(D))/(2.0*A);
    OK:=TRUE;
end QUADRATIC;
```

Si las raíces son reales se entregan en ROOT_1 y ROOT_2 y la variable OK queda con valor TRUE. En cualquier otro caso OK queda en FALSE y ROOT_1 y ROOT_2 quedan indefinidas.

Nótese el uso de la instrucción **return**. Puesto que un procedimiento no retorna un valor, si se usa la instrucción **return** (una o más veces) esta no puede ir seguida de una expresión. Cuando se llega a un **return**, el procedimiento termina y el control pasa a la unidad de donde se hizo la llamada. A diferencia de las funciones, en un procedimiento puede no haber instrucciones **return**, en este caso el procedimiento se ejecuta hasta su última instrucción y luego el control retorna al punto de llamada.

Recordemos que un parámetro **out** no es una variable propiamente tal puesto que no es posible leer (utilizar) su valor asociado. Por ejemplo, no podríamos "mejorar" el procedimiento QUADRATIC de la siguiente manera

```
procedure QUADRATIC(A,B,C:in REAL; ROOT_1,ROOT_2: out REAL;
                    OK:out BOOLEAN) is
    D:constant REAL:=B**2 - 4*A*C;
begin
    OK:= (D >= 0.0 and  A /= 0.0);
    if  OK then             -- error: NO SE PUEDE LEER UN PARAMETRO OUT!!!
        ROOT_1:=(-B+SQRT(D))/(2.0*A);
        ROOT_2:=(-B-SQRT(D))/(2.0*A);
    else
        return;
    end if;
end QUADRATIC;
```

Los parámetros formales de un subprograma pueden ser utilizados en la parte procedural como parámetros reales en llamadas a subprogramas. Por ejemplo:

```
procedure P(I:in INTEGER; J:out INTEGER; K:in out INTEGER) is
    procedure Q(L: in INTEGER; M: out INTEGER; N: in out  INTEGER) is
    begin
        N:=N+L;
        M:=N+L;
    end;
begin

    Q(I,I,I);       -- error: I es constante. No puede ser un parámetro real para
                    -- parámetros formales out o in out.
    Q(I,J,J);       -- error: J es variable especial, no puede ser leída. No puede ser
                    -- un parámetro real para parámetros formales in out.
    Q(I,J,K);       -- correcto
Q(K,J,K);       -- correcto: K no es pasada como variable, sólo se pasa su valor
                    -- al parámetro formal L.
```

```
end;
```

Entonces

```
declare
        A,B,C:INTEGER:=0;
begin
        P(A,B,C);       -- correcto
        P(A+B,5,C);     -- incorrecto: el segundo parámetro debe ser una variable
        P(3, A,B);      -- correcto.
end;
```

El mecanismo utilizado por Ada para los parámetros de modo **out** e **in out** difiere sustancialmente de pasaje de parámetros por referencia de Pascal, o el uso de punteros en C. En estos últimos casos todos los cambios que ocurren en un parámetro se van reflejando paralelamente en la variable referenciada, esto puede ocasionar problemas cuando la variable utilizada como parámetro real por referencia es, además, utilizada como una variable global dentro del subprograma. Se tendría en este caso dos (o más) identificadores para un mismo objeto (lugar físico en la memoria). Esto es parecido a tener dos o más punteros apuntando a un mismo nodo, lo que no es erróneo en sí, pero si no es bien administrado puede conducir a errores. En Ada por otro lado, también es posible usar una variable global dentro de un subprograma, la que a su vez ha sido usada como parámetro real **in out**. Pero en este caso se tiene dos objetos diferentes (dos lugares físicos de memoria). Si bien en Ada esta situación pareciera ser menos "riesgosa" es preferible evitarla.

Ejercicio:

1. Escriba un procedimiento de nombre SWAP para intercambiar los valores de dos variables reales.
2. Escriba un procedimiento SORT que ordene un arreglo de enteros.

7.4 Parámetros con nombre y por omisión.

Hasta ahora las llamadas a los subprogramas se han hecho entregando todos los parámetros en su respectivo orden. También es posible entregar los parámetros indicando su correspondiente nombre (formal). En este caso no es necesario seguir el orden en que aparecen en la especificación del subprograma. Por ejemplo, en lugar de

```
QUADRATIC(L,M,N,P,Q,STATUS);
INCREMENT(1);
```

podemos escribir

```
QUADRATIC(B => M,A => L,C => N, ROOT_1 => P,
        ROOT_2 => Q, OK => STATUS);
INCREMENT(X =>1);
```

o incluso podríamos escribir

```
INCREMENT(X => X);
```

También es posible mezclar la forma posicional de entrega de parámetros con el uso de nombres. En este caso los parámetros sin nombre deben ir al comienzo, y cuando se comienza a utilizar nombres debe continuarse así hasta el final. Por ejemplo, podemos escribir

```
QUADRATIC(L,M,N, ROOT_2 => Q,
        ROOT_1 => P, OK => STATUS);
```

El dar nombre a los parámetros al momento de la llamada tiene dos usos principales. El primero es para hacer más legible el código, por ejemplo, al escribir

```
QUADRATIC(L,M,N, ROOT_1 => P, ROOT_2 => Q, OK => STATUS);
```

podemos inferir por el nombre del procedimiento y los nombres de los parámetros que en P y Q están las raíces de la ecuación cuadrática.

El segundo uso dice relación con los valores por omisión de los parámetros. A veces ocurre que un parámetro **in** toma generalmente un mismo valor en cada llamada. En este caso es posible dar un valor por omisión, el que será pasado al parámetro formal en el momento de la llamada, a menos que se de

explícitamente un valor. Por ejemplo, al chequear a los pasajeros que llegan por vuelo internacional a Chile, es de esperar que la mayoría de ellos sean chilenos y que, además, la mayoría viaja con fines turísticos.

```
type MOTIVO is (NEGOCIOS, TURISMO, OTROS);
type PAIS is (CHILE, CHINA, INGLATERRA, etc);

procedure CHEQUEO_DESEMBARCO (NOMBRE: in NAME; ORIGEN: in PAIS;
                                        DESTINO:in PAIS:= CHILE;
                                        MOTIVO_DE_VIAJE:in MOTIVO:=TURISMO);
```

Podríamos hacer las siguientes tipos de llamadas:

```
CHEQUEO_DESEMBARCO("J.MARTINEZ", CHINA);
CHEQUEO_DESEMBARCO("J.ESCOBAR", INGLATERRA,CHINA);
CHEQUEO_DESEMBARCO("J.MORA", CHINA, MOTIVO_DE_VIAJE => NEGOCIOS);
```

Para un buen uso de los valores por omisión, todos aquellos parámetros candidatos con esta características deben colocarse al final de la lista de parámetros en la especificación del subprograma. De esta forma es posible colocar todos aquellos obligatorios en su respectivo orden al comienzo de la llamada. Luego vienen los parámetros con valores por omisión, los cuales pueden no estar presentes en su totalidad. Si uno de ellos no está presente, y no es el último, los demás parámetros deben ser dados con su respectivo nombre.

El valor por omisión puede no ser constante, puede ser en general una expresión del tipo correspondiente, la cual es evaluada cada vez que se realiza una llamada al subprograma. En nuestro ejemplo, el parámetro ORIGEN podría tener como valor por omisión ORIGEN_VUELO(Nro_VUELO), en este caso, si no se da un valor explícito el parámetro tomará el valor entregado por la función ORIGEN_VUELO, la que debería entregar un valor de tipo PAIS.

Ejercicio:

1. Escriba una función de nombre ADD que suma dos valores enteros y que toma el entero 1 como valor por omisión para el segundo parámetro. ¿De cuántas formas diferentes es posible llamar a la función ADD para que entregue N+1, donde N es el primer parámetro?

2. ¿Qué ocurrirá si todos los parámetros tienen valores por omisión y se hace uso de todos ellos?

7.5 Sobrecarga

Ya hemos visto que es posible definir nuevos significados a los operadores predefinidos del lenguaje. De hecho esta "sobrecarga" semántica se extiende a todos los subprogramas en general.

Un programa sobrecargará a uno definido con anterioridad en la medida que sea lo suficientemente diferente. Por otro lado, si el orden y los tipos de los parámetros y el resultado (para las funciones) es el mismo, entonces en lugar de sobrecarga (overloading) habrá ocultamiento (hiding). Obviamente, un procedimiento no puede ocultar una función ni una función a un procedimiento. Obsérvese que los nombres y modos de los parámetros, y la presencia o ausencia de valores por omisión no son relevantes al momento de determinar si existe sobrecarga u ocultamiento. Es posible declarar dos o más subprogramas sobrecargados en un misma parte declarativa.

Los subprogramas y los literales de enumeración pueden sobrecargarse unos a otros. De hecho un literal de enumeración es formalmente una función sin parámetros con resultado del tipo de enumeración. (Por ejemplo, el literal de enumeración SUN formalmente es una función de nombre SUN que no tiene parámetros y que entrega un valor constante del tipo DAY).

Existen dos tipos de identificadores: los sobrecargables y los no-sobrecargables. En cualquier punto de un programa Ada un identificador hace referencia a un (y sólo un) objeto no-sobrecargable o a uno o más objetos sobrecargables. La declaración de un identificador de un tipo oculta las posibles definiciones previas del otro tipo y no pueden aparecer en una misma parte declarativa.

7.6 Declaraciones, ámbito (scope) y visibilidad

Ya hemos dicho que a veces es necesario entregar la especificación de un subprograma sin su cuerpo. (Recuérdese que el cuerpo incluye la especificación.) Un ejemplo concreto en que esto es necesario se presenta cuando hay recursividad mutua entre subprogramas. Supongamos que queremos declarar dos procedimientos F y G donde cada uno llama al otro. Debido a la regla de la "elaboración lineal de declaraciones" no podemos escribir la llamada a F en el cuerpo de G sin antes haber declarado F y viceversa. Claramente esto es imposible de realizar si escribimos los cuerpos, porque necesariamente uno de ellos habrá de ir segundo. Sin embrago, podemos escribir.

```
procedure F(…);          -- declaración de F
```

```
procedure G(…) is        -- cuerpo de G
begin
        F(…);
end G;

procedure F(…) is        -- cuerpo de F repite
begin                    -- su especificación
        G(…);
end F;
```

y todo funcionará correctamente.

Si la especificación se repite debe haber una total correspondencia. Técnicamente, diremos que las dos especificaciones se corresponden. Pueden haber pequeñas variaciones, por ejemplo los valores por omisión (que deben ser escritos dos veces) no afectan la correspondencia puesto que son evaluadas sólo cuando se llama al subprograma.

A veces, para lograr mayor claridad en el código, es conveniente escribir todas las declaraciones de los subprogramas juntas para que actúen como un sumario; y a continuación se escriben todos los cuerpos.

Los cuerpos de los subprogramas y las demás declaraciones no pueden mezclarse en forma arbitraria. Los cuerpos deben estar a continuación de cualquier otra declaración. De esta forma se evita que las declaraciones "pequeñas" se pierdan entre los cuerpos de los subprogramas.

Puesto que los subprogramas se escriben en las partes declarativas y a su vez poseen partes declarativas, es posible anidar subprogramas. Las normas de ocultamiento explicadas anteriormente para los bloques también rigen para los subprogramas. Consideremos.

```
procedure P is
        I:INTEGER:=0;

        procedure Q is
                K:INTEGER:=I;
                I: INTEGER;
                J: INTEGER;
        begin
                . . .
        end Q;
        . . .
end P;
```

Como ya hemos visto la variable I interna "oculta" a la externa, sin embargo, la primera sigue existiendo y es posible utilizarla mediante la notación punto, usando como prefijo el nombre del subprograma donde está declarada la variable. En este caso el nombre completo de la variable I externa es P.I, y podríamos, por ejemplo, inicializar J escribiendo

```
J:INTEGER:=P.I;
```

El nombre completo de la variable I interna es P.Q.I, y podría ser referenciada de esa manera para, por ejemplo, explicitar su naturaleza.

(Recuérdese que los bloques también pueden tener nombres y utilizando la notación punto es posible referenciar cualquier variable que haya sido ocultada por una redefinición. Lo mismo puede hacerse con las iteraciones.)

Como hemos visto los subprogramas pueden alterar variables globales y de este modo generar efectos colaterales. (Un efecto colateral es una consecuencia de la llamada a un subprograma en su entorno y que no está relacionada con el mecanismo de parámetros.) En general se considera que los efectos colaterales son una mala práctica de programación, especialmente en el caso de las funciones.

8. Estructura general

En las secciones anteriores han sido descritas las características de Ada a pequeña escala. Lo visto del lenguaje hasta el momento difiere poco de otros lenguajes tradicionales, a pesar que Ada ofrece mayor funcionalidad en muchas áreas. Ahora comenzaremos a estudiar aspectos del lenguaje relacionados con la abstracción de datos y la programación en gran escala. En concreto hablaremos de paquetes (que es la principal característica de Ada) y de compilación separada.

8.1. Paquetes

Uno de los mayores problemas con los lenguajes estructurados como Algol, Pascal o C, es que no tienen un control adecuado para el ocultamiento de la información. Por ejemplo, supongamos que tenemos una pila representada por un arreglo y una variable que sirve de índice para el elemento en el tope, un procedimiento PUSH para agregar un elemento y una función POP para removerlo. Podríamos escribir

```
MAX: constant INTEGER:= 100 ;
S: array (1 .. MAX) of INTEGER;
TOP: INTEGER range 0 .. MAX;
```

para representar la pila y luego declarar

```
procedure PUSH(X: INTEGER) is
begin
        TOP:=TOP + 1;
        S(TOP):= X;
end PUSH;

function POP return INTEGER is
begin
        TOP:=TOP - 1;
        return S(TOP+1);
end POP;
```

En un lenguaje estructurado normal no hay forma de accesar los subprogramas PUSH y POP sin tener a la vez acceso directo a las variables S y TOP. En consecuencia no es posible obligar al uso de un protocolo correcto y así evitar el tener que conocer la forma como está implementada la pila.

Ada supera este problema mediante el uso de paquetes, los que permiten colocar "una pared" alrededor de un grupo de declaraciones y permiten el acceso sólo a aquellas que por definición son visibles. De hecho un paquete se divide en dos partes: la especificación que entrega la interfaz con el mundo externo, y el cuerpo en que se detallan los detalles ocultos (privados).

Usando paquetes, el ejemplo anterior podría escribirse como sigue

```
package STACK is                                -- especificación
        procedure PUSH(X:INTEGER);
        function POP return INTEGER;
end STACK;

package body STACK is                           -- cuerpo
        MAX: constant INTEGER:= 100;
        S: array (1 .. MAX) of INTEGER;
        TOP:INTEGER range 0 .. MAX;

        procedure PUSH(X:INTEGER) is
        begin
                TOP := TOP + 1;
                S(TOP):=X;
        end PUSH;
        function POP return INTEGER is
        begin
                TOP := TOP - 1;
                return S(TOP +1);
        end POP;
begin                                           -- inicialización
        TOP:=0;
end STACK;
```

La especificación de un paquete comienza con la palabra reservada **package**, el identificador del paquete y la palabra **is**. Luego vienen las declaraciones de las entidades que son visibles. Se finaliza con la palabra **end**, el identificador (opcional) y el punto y coma final. En el ejemplo sólo se han declarado los dos subprogramas PUSH y POP.

El cuerpo también comienza con la palabra **package**, pero seguida de **body**, el identificador y la palabra **is**. Luego viene una parte declarativa normal, **begin** , secuencia de instrucciones, **end**, el identificador (opcional) y un punto y coma.

En el ejemplo la parte declarativa del cuerpo contiene las variables usadas para representar la pila y los cuerpos de PUSH y POP. La secuencia de comandos entre **begin** y **end** es ejecutada cuando se declara el paquete y puede ser usada para inicializaciones, si estas no son requeridas el **begin** puede ser omitido. De hecho en el ejemplo hubiese sido más adecuado inicializar la variable TOP en la forma por nosotros ya conocida: TOP: INTEGER RANGE 0 ..MAX:= 0;

Nótese que un paquete es declarado y por lo tanto es sólo un ítem más en una parte declarativa de un subprograma, bloque u otro paquete, a menos que sea una unidad de biblioteca en cuyo caso no estará anidado.

No se puede colocar un cuerpo en la especificación de un paquete. Además, si la especificación de un paquete contiene la especificación de un subprograma, entonces el cuerpo del paquete debe necesariamente contener el cuerpo del subprograma. Podríamos conceptualizar a la declaración y al cuerpo de un paquete como una gran parte declarativa con algunos elementos visibles. Pero sin embargo, es posible declarar el cuerpo de un subprograma en el cuerpo del paquete, sin hacerlo en su declaración. Dicho subprograma será local y sólo podrá ser usado por otros subprogramas del paquete (subprogramas visibles o locales) o en la sección de inicialización.

La elaboración de un paquete consiste simplemente en la elaboración de sus declaraciones internas y la ejecución de la secuencia de inicialización (si existe). El paquete existe hasta el final del ámbito (scope) en el que fue declarado.

Un paquete puede ser declarado en cualquier parte declarativa (en un bloque, un subprograma o en otro bloque). Si la especificación de un paquete se realiza dentro de la especificación de otro paquete (al igual que con los subprogramas), el cuerpo del primero debe ser declarado en el cuerpo del segundo. Aparte de que en la especificación de un bloque no puede haber cuerpos, no existe ninguna otra restricción y es posible hacer cualquiera de las declaraciones vistas hasta ahora.

El paquete en sí tiene un nombre y las entidades ubicadas en su parte visible pueden ser conceptualizadas como sus componentes. Por lo tanto la manera más obvia de acceder a estos componentes es utilizando la notación punto. Por ejemplo:

```
declare
        package STACK is                                    . . .
          . . .
        end STACK;
        package body STACK is
          . . .
        end STACK;
begin
          . . .
        STACK.PUSH(M);
          . . .
        N:=STACK.POP;
end;
```

Dentro del paquete es posible accesar PUSH y POP directamente (pues son “objetos locales” al paquete), al igual que MAX, S y TOP. Pero esta últimas de NINGUNA FORMA son accesibles desde afuera del paquete. Son datos “protegidos”, están “encapsulados” y en un cierto sentido hemos creado un tipo abstracto de datos para representar una pila.

Lógicamente, podemos evitar escribir reiteradamente el nombre del paquete con al notación punto si hacemos uso del la cláusula **use**, la cual podría ir inmediatamente a continuación de la declaración del paquete o en cualquier otra parte declarativa donde este sea visible. Por ejemplo:

```
declare
        use STACK;
begin
          . . .
        PUSH(M)
          . . .
        N:=POP;
          . . .
end;
```

Es posible declarar más de un paquete en una misma parte declarativa. En general, deberíamos escribir primero todas las especificaciones y luego todos los cuerpos o alternadamente las especificaciones y

luego los cuerpos. Es decir, spec A, spec B, body A, body B, o spec A, body A, spec B, body B. Las reglas que se deben cumplir son bastante simples:

1. elaboración lineal de declaraciones.
2. la especificación debe preceder al cuerpo para cada paquete (o subprograma).
3. los itemes pequeños deberían generalmente preceder a los mayores.

Por supuesto, la parte visible de un paquete no sólo puede contener subprogramas. De hecho, un caso importante es cuando no contiene subprogramas, sino grupos de variables, constantes y tipos relacionados. Este tipo de paquetes no necesita un cuerpo, no provee ningún tipo de ocultamiento de información, sólo sirve para agrupar los objetos relacionados.

Como ejemplo podemos escribir un paquete que contiene el tipo DAY y algunas constantes relacionadas.

```
package DIURNAL is
        type DAY is (MON,TUE,WED,THU,FRI,SAT,SUN);
        subtype WEEKDAY is DAY range MON .. FRI;
        TOMORROW: constant array (DAY) of DAY:=
                        (TUE,WED,THU,FRI,SAT,SUN,MON);
        NEXT_WORK_DAY: constant array(WEEKDAY) of WEEKDAY:=
                        (TUE,WED,THU,FRI,MON);
end DIURNAL;
```

Un subprograma no puede ser llamado durante la elaboración de una parte declarativa si su cuerpo aún no a aparecido. (Nótese que esto no dificulta la recursión mutua, puesto que en este caso la llamada sólo ocurre cuando se ejecuta la secuencia de instrucciones.) Esto impide que sean usados para dar valores iniciales. Por ejemplo

```
function A return INTEGER;
I: INTEGER:= A;
```

es ilegal, y debería provocar una excepción PROGRAM_ERROR.

Esta regla también se aplica a los subprogramas dentro de los paquetes. No podemos llamar a un subprograma desde fuera de un paquete hasta que el cuerpo del paquete ha sido elaborado.

Ejercicio: Escriba un paquete COMPLEX_NUMBERS que haga visible

1. el tipo COMPLEX
2. la contante imaginaria I = $\sqrt{-1}$
3. las funciones +, - , * y / que actúan sobre los complejos.

8.2. Unidades de biblioteca

En el pasado muchos lenguajes han ignorado el hecho que los programas son divididos en partes que se compilan separadamente y que luego se unen en un todo. Ada reconoce este hecho y provee dos mecanismos: uno top-down y otro bottom-up.

El primero es apropiado para el desarrollo de un programa grande fuertemente cohesionado, el que, sin embargo, por alguna razón es dividido en módulos que pueden compilarse separadamente, con posterioridad al programa general. El segundo mecanismo está orientado a escribir bibliotecas con módulos de propósito general, los que son compilados antes de los programas que se espera harán uso de ellos.

Una unidad de biblioteca es la especificación de un subprograma o un paquete. A los correspondientes cuerpos se les denomina unidades secundarias. Estas unidades pueden ser compiladas en forma separada, o por conveniencia es posible compilar varias de ellas juntas. Es decir, podemos compilar las especificación y el cuerpo de un paquete juntos, pero como veremos más adelante también es posible realizar la compilación separadamente. Como hemos visto el cuerpo de un subprograma es suficiente para definirlo en su totalidad, por lo que es clasificado como una unidad de biblioteca y no una unidad secundaria.

Cuando se compila una unidad va a parar a una biblioteca. Lógicamente, habrá varias bibliotecas de acuerdo al usuario, proyecto, área de aplicación, etc. (Aquí no nos preocuparemos de la creación y manipulación de bibliotecas, sino de su aplicación y uso desde la perspectiva del "programador usuario" de ellas.) Una vez que está en una biblioteca, la unidad puede ser usada por cualquier otra unidad compilada, pero la unidad que llama (que usa) debe indicar explícitamente la dependencia con una cláusula **with**.

Veamos un ejemplo sencillo. Supongamos que compilamos el paquete STACK, el que no depende de ninguna otra unidad y por lo tanto no requiere de cláusulas **with**. Compilaremos la especificación y el cuerpo juntos. Luego el texto a compilar será

```
package STACK is
  . . .
end STACK;

package body STACK is
  . . .
end STACK;
```

Supongamos que escribimos el procedimiento MAIN que hace uso del paquete STACK. (El procedimiento MAIN es el programa principal en el sentido común del término.) El texto a compilar sería

```
with STACK;
procedure MAIN is
        use STACK
        M,N:INTEGER;
begin
        . . .
        PUSH(M);
        . . .
        N:=POP;
        . . .
end MAIN;
```

La cláusula **with** va antes de la unidad involucrada, de esta manera la dependencia entre unidades queda clara. Un cláusula **with** no puede estar anidada en ámbitos (scope) internos.

Si una unidad depende de varias otras unidades, éstas pueden ir todas juntas en un sola cláusula **with** o separadas. Por ejemplo, podemos escribir

```
with STACK, DIURNAL;
procedure MAIN is
    . . .
```

o, lo que es equivalente

```
with STACK;
with DIURNAL;
procedure MAIN is
            . . .
```

Por conveniencia colocaremos la cláusula **use** inmediatamente después de cada **with**. Entonces, si escribimos

```
with STACK; use STACK;
procedure  MAIN is
                . . .
```

los procedimientos PUSH y POP estarán directamente accesibles dentro del procedimiento, sin necesidad de usar la notación punto.

Sólo las dependencias directas se deben indicar con la cláusula **with**. Por ejemplo, si el paquete P usa las facilidades del paquete Q, y este a su vez usa un paquete R, entonces la cláusula **with** de P sólo debe mencionar a Q. El usuario de P no debe preocuparse por R.

Otro punto importante es que una cláusula **with** aplicada a la declaración de un bloque o subprograma también es válida para el respectivo cuerpo, por lo que no necesita ser repetida. Por supuesto, un cuerpo puede tener dependencias adicionales, las que deberán indicarse con otras cláusulas **with** válidas sólo para el cuerpo. Estas últimas no deberían indicarse para las especificación, debido a que con esto se reduce la independencia de la declaración.

Si la especificación y cuerpo de un paquete son compilados en forma separada, entonces el cuerpo siempre debe ser compilado después de la declaración. Se dice que “el cuerpo depende de la especificación”. Sin embargo, cualquier otra unidad que hace uso del paquete sólo depende de la especificación y no del cuerpo. Si el cuerpo es modificado en cualquier grado que no afecte la especificación respectiva, las unidades que hacer uso del paquete (es decir, que dependen de él) no necesitarán ser recompiladas. La capacidad de compilación separada de declaraciones y cuerpos debería incidir en la simplificación de la mantención de los programas.

La regla general de compilación es bastante simple: una unidad debe ser compilada después de todas las unidades de las cuales depende. En consecuencia, si una es modificada y recompilada, entonces todas sus unidades dependientes deben recompilarse. Se puede usar cualquier orden en la recompilación que sea consistente con la regla de dependencia.

Existe un paquete que no necesita ser mencionado en una cláusula **with**. Este es el paquete STANDARD que contiene las declaraciones de todas los tipos predefinidos tales como INTEGER y

BOOLEAN. En el está declarado un paquete de nombre ASCII que contiene constantes los caracteres de control tales como CR y LF.

Otra regla importante es que las unidades de biblioteca no pueden tener nombres repetidos, es decir, no pueden ser sobrecargadas. Además, no pueden ser operadores.

La especificación y el cuerpo de un paquete forman una sola parte declarativa. Por lo tanto, si se declara una variable X en la especificación, entonces no se puede ser redeclarada en el cuerpo (excepto, lógicamente, si se declara en una región interna como, por ejemplo, en un subprograma).

Ejercicio. El paquete D y los subprogramas P, Q y MAIN tienen las siguientes cláusulas **with**

especificación de D	no tiene cláusulas **with**
cuerpo de D	**with** P,Q;
subprograma P	no tiene cláusulas **with**
subprograma Q	no tiene cláusulas **with**
subprograma MAIN	**with** D;

Se pide que dibuje un grafo mostrando las dependencias entre las unidades. Indique a lo menos 4 ordenes posibles de compilación. Indique qué unidades pueden ser recompiladas sin que eso implique la recompilación de las demás.

9. Tipos Privados

De acuerdo a lo que hemos visto hasta el momento una forma adecuada de "encapsular" la estructura y operaciones sobre números complejos sería mediante el siguiente paquete.

```
package COMPLEX_NUMBERS is
        type COMPLEX is
                record
                    RL, IM: REAL;
                end record;
        I : constant COMPLEX := (0.0, 1.0);
        function "+" (X,Y:COMPLEX) return COMPLEX;
        function "-" (X,Y:COMPLEX) return COMPLEX;
        function "*" (X,Y:COMPLEX) return COMPLEX;
        function "/" (X,Y:COMPLEX) return COMPLEX;
end;
```

El problema con esta formulación es que el usuario (el programador que utiliza de este paquete) puede hacer uso del hecho que los números complejos están representados en forma cartesiana. En lugar de usar siempre el operador complejo +, podría escribir cosas como

```
C.MI:= C.MI + 1.0;
```

en lugar de

```
C:= C + I;
```

De hecho, usando este paquete, el usuario debería hacer uso de la representación para la construcción valores complejos. Por ejemplo

```
declare
        X:COMPLEX;
begin
        X.RL:= 5.0;
        X.IM:=  1.5;
end;
```

Sería deseable no tener que depender de la representación ni conocerla, para de ese modo poder cambiar la representación, por ejemplo, a coordenadas polares sin que esto repercuta en los programas que usan estos complejos. Esto puede hacerse con los tipos privados. Por ejemplo

```
package COMPLEX_NUMBERS is
        type COMPLEX is private;
        I:constant COMPLEX;
        function "+" (X,Y:COMPLEX) return COMPLEX;
        function "-" (X,Y:COMPLEX) return COMPLEX;
        function "*" (X,Y:COMPLEX) return COMPLEX;
        function "/" (X,Y:COMPLEX) return COMPLEX;
        function CONS (R,I:REAL) return COMPLEX;
        function RL_PART (X:COMPLEX) return REAL;
        function IM_PART (X:COMPLEX) return REAL;
private
        type COMPLEX is
                record
                        RL, IM: REAL;
                end record;
        I : constant COMPLEX:= (0.0, 1.0);
end;
```

La parte de la especificación del paquete que se encuentra antes de la palabra reservada **private** es la parte visible y da la información disponible para los usuarios del paquete. El tipo COMPLEX es declarado como privado. Esto significa que fuera del paquete nada se sabe respecto a los detalles del tipo. Las únicas operaciones disponibles son la asignación, =, y /= más aquellas agregadas por el autor del paquete como subprogramas especificados en la parte visible.

También podemos declarar constantes de un tipo privado en la parte visible, como la constante I de nuestro ejemplo. El valor inicial no puede ser declarado en la parte visible debido a que los detalles del tipo

aún no son conocidos. Por lo tanto, sólo indicamos que I es una constante. A estas constantes de les denomina **constantes diferidas**.

Después de **private** debemos dar los detalles de los tipos declarados como privados y dar los valores iniciales a las constantes diferidas.

Un tipo privado puede ser implementado de cualquier forma que sea consistente con las operaciones visibles al usuario. Puede ser un registro, como en nuestro ejemplo; o podría ser un arreglo, un tipo de enumeración u otra forma válida; podría incluso ser declarado en términos de otro tipo privado. En nuestro caso, es obvio que la forma más natural de representar un complejo es por medio de un registro; pero podríamos igualmente haber usado un arreglo de dos elementos, tal como

```
type COMPLEX is array (1 .. 2) of REAL;
```

Habiendo declarado los detalles de un tipo privado, estamos en posición de usarlo y declarar constantes y darles valores iniciales.

Debe notarse que junto con las funciones +, -, * y / hemos agregado CONS para crear números complejos a partir de sus componentes real e imaginario; y RL_PART y IM_PART para entregar estos componentes. Lógicamente, el hecho que CONS, RL_PART y IM_PART correspondan a nuestra visión de los números complejos en su notación cartesiana no impide que podamos implementarlos en su forma polar.

El cuerpo del paquete sería como sigue

```
package body COMPLEX_NUMBERS is
        function "+"  (X,Y:COMPLEX) return COMPLEX is
        begin
                return (X.RL + Y.RL, X.IM + Y.IM);
        end "+";
        --- las demás funciones (-, * y / ) se implementan en forma similar

        function CONS(R,I: REAL) return COMPLEX is
        begin
                return (R,I);
        end CONS;

        function RL_PART (X:COMPLEX) return REAL is
        begin
                return X.RL;
        end RL_PART;

        function IM_PART (X:COMPLEX) return REAL is
        begin
                return X.IM;
        end IM_PART;
end COMPLEX_NUMBERS;
```

El paquete COMPLEX_NUMBERS podría ser usado de la manera siguiente

```
declare
        use COMPLEX_NUMBERS;
        C,D: COMPLEX
        R,S: REAL;
begin
        C:= CONS(1.5, -6.0);
        D:= C + I;                      -- suma de complejos
        R:= RL_PART(D) + 6.0;   --  suma de reales
        . . .
end;
```

Fuera del paquete podemos declarar variables y constantes del tipo complejo de la manera usual. Nótese el uso de la función CONS para crear complejos.

No es posible combinar operaciones entre números complejos y reales. Por ejemplo, sería incorrecto

```
C:= 2.0 * C;
```

sino que deberíamos escribir

```
C:= CONS(2.0, 0.0) * C;
```

Si esto resultase muy tedioso se podría agregar otros operadores (sobrecargados) para permitir las operaciones mezcladas.

Supongamos que por alguna razón hemos decidido representar los números complejos en coordenadas polares. La parte visible del paquete no sufrirá cambios, pero la parte privada quedaría

```
private
        PI: constant:= 3.1415926536;
        type COMPLEX is
                record
                        R:REAL;
                        THETA: REAL range 0.0 .. 2.0 * PI;
                end record;
        I : constant COMPLEX := (1.0, 0.5*PI);
end;
```

El cuerpo del paquete COMPLEX_NUMBERS necesita ser totalmente reescrito. Algunas funciones serán más simples y otras más complejas. Sin embargo, ***debido a que la parte visible no ha cambiado, quienes hacen uso del paquete no se ven afectados; estamos seguros de eso puesto que no es posible escribir código que dependa de los detalles de implementación de un tipo de dato privado***. Sin embargo, las unidades dependientes del paquete COMPLEX_NUMBERS deberán ser recompiladas de acuerdo a las reglas generales de dependencia ya estudiadas. Esto puede parecer contradictorio, pero debemos recordar que el compilador necesita información sobre la parte privada para asignar espacio para los objetos del tipo privado declarados en las unidades usuarias. Si se cambia la parte privada el tamaño de los objetos podría cambiar.

Finalmente, notemos que entre la declaración de un tipo privado y su posterior declaración en extenso, el tipo está en un curioso estado de definición a medias. Debido a esto su uso está sujeto a serias restricciones; sólo puede ser usado para declarar constantes diferidas, otros tipos y subtipos y especificaciones de subprogramas. No puede ser usado para declarar variables. Podríamos escribir

```
type COMPLEX_ARRAY is array (INTEGER range <>) of COMPLEX;
```

y luego

```
C: constant COMPLEX_ARRAY(1 .. 10);
```

Pero sólo después de la declaración en extenso del tipo podemos declarar variables de los tipos COMPLEX o COMPLEX_ARRAY.

Sin embargo, podemos declarar especificaciones de subprogramas con parámetros de tipo COMPLEX o COMPLEX_ARRAY e incluso podemos asignarles expresiones por omisión. Estas expresiones pueden hacer uso de constantes y funciones. Esto se permite porque una expresión por omisión sólo se evalúa cuando efectivamente se llama a un subprograma y esto sólo puede ocurrir una vez que el cuerpo ha sido declarado, lo cual ocurre - obligatoriamente - después de haber definido el tipo en extenso.

Ejercicio:

1. Escriba funciones "*" adicionales para permitir la multiplicación mixta entre complejos y reales.

2. Complete el paquete RATIONAL_NUMBERS cuya parte visible es

```
package RATIONAL_NUMBERS is
        type RATIONAL is private;
        function "+" (X:RATIONAL) return RATIONAL; --   + unario
        function "-" (X:RATIONAL) return RATIONAL;  --   - unario
        function "+" (X,Y:RATIONAL) return RATIONAL;
        function "-" (X,Y:RATIONAL) return RATIONAL;
        function "*" (X,Y:RATIONAL) return RATIONAL;
        function "/" (X,Y:RATIONAL) return RATIONAL;

        function "/" (X:INTEGER;Y:POSITIVE) return RATIONAL;
        function NUMERATOR (X:RATIONAL) return INTEGER;
        function DENOMINATOR (X:RATIONAL) return POSITIVE;
private
        . . .
end;
```

Un número racional es un número de la forma N/D, donde N es un entero y D es un entero positivo. Para que la igualdad predefinida trabaje es esencial que los número racionales sean reducidos a su mínima expresión. Por ejemplo, los racionales 12/2, 24/4, etc deberían expresarse como 6/1.

3. ¿Por qué

function "/" (X:INTEGER;Y:POSITIVE) **return** RATIONAL;

no se sobrepone a la división entera predefinida?

10. Excepciones

Una excepción es una situación que requiere de un tratamiento especial que escapa al funcionamiento normal de un programa (o parte de él). En los capítulos anteriores varias veces se ha indicado que si ocurre un error durante la ejecución de un programa se origina una excepción (generalmente CONSTRAIN _ERROR). En este capítulo se describirá el mecanismo de manejo de excepciones.

Algunas de las excepciones predefinidas dentro del lenguaje Ada son:

CONSTRAIN_ERROR: generalmente indica que algo se ha salido de rango.

NUMERIC_ERROR: esto ocurre cuando algo erróneo pasó en los cálculos aritméticos, por ejemplo, el intentar dividir por cero.

PROGRAM_ERROR: ocurre si se intenta violar de alguna manera el control de ejecución, por ejemplo, al llamar a un subprograma cuyo cuerpo (**body**) todavía no ha sido construido.

STORAGE_ERROR: ocurre al salirse del espacio de memoria, por ejemplo, se llama a una función recursiva FACTORIAL con un parámetro muy grande.

10.1. Manejo de excepciones

Si sabemos que una excepción puede ocurrir en alguna parte de nuestro programa, podemos escribir un "manejador de excepciones" para tratar la situación. Por ejemplo, supongamos que escribimos

```
begin
        -- secuencia de comandos
exception
        when CONSTRAIN_ERROR => -- hacer algo
        . . .
end;
```

Si se origina una excepción CONSTRAIN_ERROR mientras se ejecuta la secuencia de comandos entre **begin** y **excepction**, entonces el flujo de control es interrumpido e inmediatamente transferido a la secuencia de comandos que sigue a =>. A la cláusula que comienza con **when** se le conoce como "manejador de excepción" (exception handler).

Un ejemplo trivial sería el determinar el valor de una variable TOMORROW a partir de TODAY (ambas de tipo DAY)

```
begin
        TOMORROW:= DAY´SUCC(TODAY);
exception
        when CONSTRAIN_ERROR =>
                TOMORROW:= DAY´FIRST;
end;
```

Si **TODAY** es **DAY´LAST** (es decir, **SUN**) cuando se intente evaluar **DAY´SUCC(TODAY)** surgirá la excepción **CONSTRAIN_ERROR**. El control pasará entonces al manejador de **CONSTRAIN_ERROR** y se evaluará la expresión **DAY´FIRST**, cuyo valor será entregado como resultado.

En realidad este no es un buen ejemplo puesto que las excepciones deberían ser usadas en casos con baja probabilidad de ocurrencia. Puesto que el 14% de los días son domingo sería más adecuado el siguiente código.

```
if TODAY = DAY´LAST then
        TOMORROW:= DAY´FIRST;
else
        TOMORROW:= DAY´SUCC(TODAY);
end if;
```

sin embargo, es un ejemplo sencillo que ilustra el mecanismo involucrado.

Es importante recalcar que el control nunca vuelve a la unidad donde apareció la excepción. La secuencia de comandos que sigue a **=>** reemplaza al resto de la unidad y completa la ejecución de ésta.

Entre **exception** y **end** es posible escribir varios manejadores de excepciones,

```
begin
        -- secuencia de comandos
```

```
exception
        when NUMERIC_ERROR|CONSTRAIN_ERROR =>
                PUT("Numeric or Constrain error ocurred");
        when STORAGE_ERROR =>
                PUT("Ran out of space");
        . . .
        when others =>
                PUT ("Something else went wrong");
        . . .
end;
```

En el ejemplo se envía un mensaje de acuerdo al tipo de excepción. Nótese la similitud con el comando case. Cada **when** es seguido por uno o más nombres de excepciones separados por barras verticales. Como es usual, podemos escribir **others**; con las restricciones y significado análogos al comando **case**.

Los manejadores de excepciones pueden aparecer al final de un bloque, cuerpo de subprograma, cuerpo de paquete o cuerpo de una tarea, y tienen acceso a todas las entidades declaradas en la unidad. Los ejemplos anteriores mostraban un bloque degenerado que no contiene una parte declarativa. Una versión correcta para determinar TOMORROW como una función sería

```
function TOMORROW(TODAY:DAY) return DAY is
begin
        return DAY´SUCC(TODAY);
exception
        when CONSTRAIN_ERROR =>
                return  DAY´FIRST;
end TOMORROW;
```

Es importante recalcar que el control nunca vuelve a la unidad donde se originó la excepción. La secuencia de comandos que sigue a => reemplaza la que queda de la unidad en cuestión y así completa su ejecución. Por lo tanto, un manejador de excepciones dentro de una función debería contener una instrucción **return** para entregar un resultado de "emergencia".

Una instrucción **goto** no puede transferir en control desde una unidad a uno de sus manejadores o viceversa, o desde un manejador a otro. Sin embargo, aparte de esta restricción, las instrucciones dentro de un manejador pueden ser tan complejas como se requiera.

Un manejador declarado al final del cuerpo de un paquete se aplica sólo a la secuencia de inicialización del paquete y no a los subprogramas del paquete. Estos últimos deberían tener sus propios manejadores.

¿Qué ocurre si se origina una excepción para la cual la unidad involucrada no posee un manejador? La respuesta es que la excepción se propaga dinámicamente. Esto significa que la ejecución de la unidad termina y la excepción es traspasada al punto en el cual se realizó la llamada a la unidad. En el caso de un bloque se busca un manejador en la unidad que lo contenga.

En el caso de un subprograma, la llamada termina y se busca un manejador en la unidad desde la que se llamó al subprograma. Este proceso se repite hasta que se llega a una unidad que contenga el manejador apropiado o se alcanza al nivel superior, es decir, se llega al programa principal y se obtendrá un mensaje ad-hoc del ambiente de ejecución (run time environment).

Es importante entender que las excepciones se propagan dinámica y no estáticamente. Es decir, una excepción que no es manejada por un subprograma es propagada a la unidad que llama al subprograma y no a la unidad que contiene su declaración (el que puede o no ser el mismo).

Si las instrucciones de un manejador a su vez originan una excepción, la unidad es terminada y la excepción se propaga a la unidad llamadora: los manejadores no entran en loop.

10.2. Declaración y generación de excepciones

En general, no es una buena práctica el utilizar las excepciones predefinidas para detectar situaciones inusuales, porque no se garantiza que las excepciones han surgido por las situaciones que se están modelando, y no por alguna otra situación anómala.

Como ejemplo consideremos el paquete STACK de la sección 8.1. Si llamamos a PUSH cuando la pila está llena, entonces la instrucción TOP:= TOP + 1; conducirá a CONSTRAIN_ERROR. Análogamente, tendremos una excepción del mismo tipo con TOP:= TOP - 1; con la función POP. Puesto que ninguno de los subprogramas tiene manejador de excepción, esta se propagará a la unidad que los llama. Entonces, podríamos escribir

```
declare
        use STACK;
```

```
begin
        . . .
        PUSH(M);
        . . .
        N:=POP;
        . . .
exception
        when CONSTRAIN_ERROR =>
                -- ¿manipulación incorrecta de la pila?
end;
```

y el uso incorrecto de la pila hará que el control se transfiera al manejador de la excepción CONSTRAIN_ERROR. Sin embargo, no existe garantía de que la excepción se origine debido al uso incorrecto de la pila; puesto que alguna otra situación dentro del bloque y ajena al uso de la pila podría provocar una excepción CONSTRAIN_ERROR

Una solución mejor es generar una excepción especialmente declarada para indicar el uso incorrecto de la pila. Entonces, el paquete podría reescribirse de la siguiente manera

```
package STACK is
        ERROR: exception;
        procedure PUSH(X:INTEGER);
        function POP return INTEGER;
end STACK;

package body STACK is
                MAX: constant INTEGER:= 100;
                S: array (1 .. MAX) of INTEGER;
                TOP:INTEGER range 0 .. MAX;

        procedure PUSH(X:INTEGER) is
                begin
                        if TOP = MAX then
                                raise ERROR;
                        end if;
                        TOP := TOP + 1;
                        S(TOP):=X;
                end PUSH;

        function POP return INTEGER is
                begin
                        if TOP = 0 then
                                raise ERROR;
                        end if;
                        TOP := TOP - 1;
                        return S(TOP +1);
                end POP;

begin
                TOP:=0;
end STACK;
```

Una excepción se declara de manera similar a una variable y es generada explícitamente por una instrucción **raise** junto con el nombre de la excepción. Las reglas de manejo y propagación son las mismas que las de las excepciones predefinidas. Ahora podemos escribir

```
declare
        use STACK;
begin
        . . .
        PUSH(M);
        . . .
        N:=POP;
        . . .
exception
       when ERROR =>
                -- uso incorrecto de la pila.
        when others  =>
                -- alguna otra cosa errónea.
```

```
end;
```

(Nótese que si no se hubiese colocado la cláusula **use** habría sido necesario referirse a la excepción como STACK.ERROR.)

Ya sabemos como declarar una excepción y como originarla ¿Pero qué deberíamos colocar dentro del manejador de la excepción? Además de reportar que ha habido un error en el uso de la pila, deberíamos dejar la pila en un estado aceptable, sin embargo hasta ahora no hemos provisto la forma de hacer esto. Sería útil agregar un procedimiento RESET al paquete STACK. Otra cosa que sería necesario hacer es devolver todos los recursos que habían sido tomados en el bloque, para así evitar que sean retenidos inadvertidamente. Supongamos, por ejemplo, que también habíamos estado utilizando el paquete PRINTER_MANAGER que contiene entre otros procedimientos ASSIGN_PRINTER (que permiten asignar una impresora al proceso que llama al procedimiento) y RETURN_PRINTER (que devuelve la impresora al sistema).

Además, sería conveniente limpiar la pila y devolver la impresora en caso de ocurrir cualquier otra excepción. Para realizar esto lo mejor sería declarar un procedimiento CLEAN_UP. Entonces, el bloque quedaría

```
declare
        use STACK, PRINTER_MANAGER;
        MY_PRINTER: PRINTER;

        procedure CLEAN_UP is
        begin
                RESET;
                RETURN_PRINTER(MY_PRINTER);
        end;

begin
        ASSIGN_PRINTER(MY_PRINTER);
        . . .
        PUSH(M);
        . . .
        N:=POP;
        . . .
        RETURN_PRINTER(MY_PRINTER);
exception
        when ERROR  =>  PUT("STACK used incorrectly");
                        CLEAN_UP;
        when others     =>  PUT("Something else went wrong");
                        CLEAN_UP;
end;
```

(Se ha asumido que en el paquete STACK se ha declarado un procedimiento RESET.)

A veces las acciones que se deben tomar ante una excepción se deben realizar por niveles. En el ejemplo anterior se devuelve la impresora y se limpia la pila, pero probablemente se requiera que el bloque completo sea desechado. Podríamos indicar esto generando una excepción como la acción final del manejador.

```
excepcion
        when ERROR =>  PUT("Stack used incorrectly");
                       CLEAN_UP;
                        raise ANOTHER_ERROR;
when others =>
        . . .
end;
```

De este modo la excepción ANOTHER_ERROR se propagará a la unidad que contenga el bloque.

A veces es conveniente manejar una excepción y luego propagar la misma excepción. Esto puede hacerse escribiendo simplemente

```
        raise;
```

Esto es particularmente útil cuando se manejan varias excepciones con un solo manejador debido a que no es posible nombrar explícitamente la excepción que ha ocurrido. Entonces, podríamos tener

```
        . . .
        when others =>  PUT ("Something else went wrong");
                        CLEAN_UP;
```

```
                    raise;
end;
```

El ejemplo de la pila muestra un uso legítimo de excepciones. La excepción ERROR debería ocurrir muy rara vez y sería, por ende, poco conveniente chequear esta condición cada vez que se pudiera producir. Para hacer esto seguramente tendríamos que agregar un parámetro adicional (en modo **out**) de tipo BOOLEAN al procedimiento PUSH para indicar que algo no funcionó correctamente y luego chequear el valor de esta parámetro después de cada llamada. En el caso de POP el cambio sería mayor puesto que ya no podría ser una función debido al uso de un parámetro de modo **out**. Entonces, la especificación del paquete quedaría

```
package STACK is
        procedure PUSH(X: in INTEGER; B: out BOOLEAN);
        procedure POP(X: out INTEGER; B: out BOOLEAN);
end;
```

y para usarlo deberíamos escribir

```
declare
        use STACK;
        OK: BOOLEAN;
begin
        . . .
        PUSH(M,OK);
        if not OK then  . . .   end if;
        . . .
        POP(N, OK);
        if not OK then  . . .  end if;
        . . .
end;
```

Es claro que en este caso el uso de excepciones contribuye a una mejor estructuración del programa.

Finalmente, nótese que nada impide generar excepciones predefinidas. Por ejemplo en la sección 7.1 cuando se analizaba la función INNER (pág. 40) se dijo que probablemente la mejor manera de controlar que los límites de los dos parámetros fuesen los mismos era generando explícitamente la excepción CONSTRAIN_ERROR.

10.3. Ambito de validez de las excepciones

En gran parte las excepciones siguen las mismas reglas de validez que las otras entidades (variables, tipos, subprogramas). Una excepción puede ocultar o ser ocultada por otra declaración; puede hacerse visible mediante la notación punto, etc. Sin embargo, son diferentes en otros aspectos. No podemos declarar arreglos de excepciones, y no pueden ser componentes de registros, parámetros de subprogramas, etc. De hecho, son sólo etiquetas, identificadores de una situación.

Una característica importante de las excepciones es que no son creadas dinámicamente a medida que se ejecuta el programa, sino que se les debe conceptualizar como existentes a través de toda la vida del programa. Esto se relaciona con la forma en que las excepciones se propagan dinámicamente por la cadena de ejecución, y no estáticamente por la cadena determinada por las reglas de ámbito de validez. Una excepción puede ser propagada fuera de su ámbito de validez, pero sólo puede ser manejada anónimamente por **others**. Veamos el ejemplo siguiente

```
declare
        procedure P is
                X: exception;
        begin
                raise X;
        end P;
begin
        P;
exception
        when others =>
                -- Aquí se maneja la excepción X
end;
```

El procedimiento P declara y genera la excepción X, pero no la maneja. Cuando llamamos a P, la excepción X es propagada al bloque que llama a P donde es manejada anónimamente.

Incluso es posible propagar una excepción fuera de su rango de validez, donde se vuelve anónima, y luego volver a donde puede ser manejada explícitamente por su nombre. Consideremos

```
declare
        package P is
                procedure F;
                procedure H;
        end P;
        procedure G is
        begin
                P.H;
        exception
                when others=>
                        raise;
        end G,
        package body P is
                X: exception;
                procedure  F is
                begin
                        G;
                exception
                        when  X =>
                                PUT("Got it");
                end F;
                procedure H is
                begin
                        raise X,
                end H;
        end P;
begin
        P.F;
end;
```

En el bloque se declara un paquete P que contiene los procedimientos F y H; y también un procedimiento G. El bloque llama a F (que está en P), el que llama a G (que está fuera de P), el que a su vez llama a H (que está en P). El procedimiento H genera la excepción X cuyo ámbito de validez es el cuerpo de P. El procedimiento H no maneja X , entonces esta se propaga a G (que llamó a H). El procedimiento G está fuera del paquete P, es decir, la excepción X está fuera de su ámbito de validez; a pesar de ello G maneja la excepción anónimamente y la propaga. G es llamado por F, por lo tanto, X es propagada de vuelta al paquete y puede ser manejada explícitamente por F.

Otro característica de las excepciones la muestra el siguiente ejemplo de una función recursiva. A diferencia de las variables, no se obtiene una nueva excepción por cada llamada recursiva. Cada activación recursiva hace referencia a la misma excepción.

```
procedure F(N:INTEGER) is
        X: exception;
begin
        if N=0 then
                raise X;
        else
                F(N-1);
        end if;
exception
        when X =>
                PUT("Got it");
                raise;
        when others =>
                null;
end F;
```

Supongamos que ejecutamos F(4); se tendrán las llamadas recursivas F(3), F(2), F(1) y F(0). Cuando F es llamado con parámetro cero genera la excepción X, la maneja: escribe un mensaje de confirmación y la propaga. La activación recursiva que llama (en este caso F(1)) recibe la excepción y nuevamente la maneja, y así para todas las llamadas. Por lo tanto, el mensaje se escribe 5 veces y finalmente la excepción se propaga anónimamente. Obsérvese que si cada activación recursiva hubiese creado su propia (diferente) excepción el mensaje se hubiese escrito sólo una vez.

En todos los ejemplos que hemos visto las excepciones se generan en instrucciones. Sin embargo, también pueden generarse en declaraciones. Por ejemplo, la declaración

```
N:POSITIVE:=0;
```

generaría CONSTRAIN_ERROR porque el valor inicial de N no satisface la restricción 1 .. INTEGER´LAST del subtipo POSITIVE. Una excepción generada en una declaración no es manejada por un manejador (si existe) de la unidad que la contiene, sino que se propaga inmediatamente a nivel superior. Esto significa que en cualquier manejador estamos seguros que todas las declaraciones de la unidad fueron elaboradas satisfactoriamente y, por lo tanto, no hay riesgo de utilizar algo que no existe.

11. Genéricos

En este capítulo se describe el mecanismo genérico (***generic***) que permite parametrizar subprogramas y paquetes con tipos y subprogramas así como valores y objetos.

11.1. Declaraciones e Instanciaciones

Uno de los problemas con un lenguaje tipificado, como Ada, es que todos los tipos deben ser determinados en tiempo de compilación. Esto significa naturalmente que no podemos pasar tipos como parámetros en tiempo de ejecución. Pero con frecuencia llegamos a la situación en que la lógica de una pieza de programa es independiente de los tipos involucrados y por lo tanto pareciese ser innecesario el repetirla para todos los diferentes tipos para los cuales pudiésemos desear aplicarla. Un ejemplo simple está propuesto por el procedimiento SWAP.

```
procedure SWAP(X, Y: in out Real) is
        T: Real;
begin
        T:=X; X:=Y; Y:=T;
end;
```

Es claro que la lógica es independiente del tipo de los valores que están siendo intercambiados. Si también quisiésemos intercambiar enteros o booleanos podríamos escribir otros procedimientos, pero esto sería tedioso. El mecanismo genérico nos permite superar esto. Podemos declarar:

```
generic
        type ITEM is private;
procedure EXCHANGE(X,Y: in out ITEM);

procedure EXCHANGE(X,Y: in out ITEM) is
        T:ITEM;
begin
        T:=X; X:=Y; Y:=T,
end;
```

El subprograma EXCHANGE es un subprograma genérico y actúa como una plantilla (template). La especificación del subprograma es precedida por la parte formal genérica consistente de la palabra reservada **generic** seguida de una lista (posiblemente vacía) de parámetros genéricos formales. El cuerpo del subprograma está escrito igual que siempre pero hay que notar que, en caso de un subprograma genérico, debemos escribir la especificación y el cuerpo separadamente.

El procedimiento genérico no puede ser llamado directamente, pero a partir de él podemos crear un procedimiento efectivo mediante el mecanismo conocido como ***instanciación genérica***. Por ejemplo, podríamos escribir:

```
procedure SWAP is new EXCHANGE(REAL);
```

En esta declaración se indica que SWAP debe ser obtenido de la plantilla descrita por EXCHANGE. Los parámetros genéricos reales son proporcionados en una lista de parámetros en la forma usual. El parámetro real (en el ejemplo es el tipo REAL) corresponde al parámetro formal ITEM.

De modo que ahora hemos creado el procedimiento SWAP actuando sobre el tipo REAL y podemos por consiguiente llamarlo en la forma usual. Podemos crear nuevas instanciaciones como:

```
procedure SWAP is new EXCHANGE(INTEGER);
procedure SWAP is new EXCHANGE(DATE);
```

y muchas más. Nótese que estamos creando nuevas sobrecargas de SWAP, las cuales pueden ser distinguidas por sus tipos de parámetros del mismo modo que si las hubiésemos escrito en detalle.

Superficialmente, puede parecer que el mecanismo genérico es simplemente una substitución de texto y en efecto, en este sencillo ejemplo, el comportamiento es el mismo. Sin embargo la diferencia importante se relaciona con el significado de los identificadores utilizados en el cuerpo genérico, y que no son ni parámetros ni objetos locales. Tales identificadores no locales poseen significados de acuerdo a donde fue declarado el cuerpo genérico y no donde éste es instanciado. Si se usara la simple substitución de texto, los identificadores no locales podrían, por supuesto, tomar su significado en el punto de instanciación y esto podría producir resultados distintos de los esperados.

Así como podemos escribir subprogramas genéricos también podemos tener paquetes genéricos. Un ejemplo simple de esto es entregado por el paquete STACK. El problema con ese paquete, es que sólo trabaja sobre tipos INTEGER aunque, por supuesto, la misma lógica se aplica sin distinción del tipo de los

valores manipulados. También podemos aprovechar la oportunidad para hacer de MAX un parámetro de la misma forma, de tal modo que no estamos atados a un límite arbitrario de 100. Escribimos:

```
generic
        MAX:POSITIVE;
        type ITEM is private;
package STACK is
        procedure PUSH(X: ITEM);
        function POP return ITEM;
end STACK;

package body STACK is
        S: array (1..MAX) of ITEM;
        TOP: INTEGER  range 0..MAX;
        -- el resto como antes, pero  donde aparecía  INTEGER
        -- ahora aparece ITEM
end STACK;
```

Ahora podemos crear y usar una pila de un tipo y un tamaño particular mediante la instanciación del paquete genérico de la siguiente forma:

```
declare
        package MY_STACK is new STACK(100, REAL);
        use MY_STACK;
begin
        ....
        PUSH(X);
        ....
        Y:=POP;
        ....
end;
```

El paquete MY_STACK, que es el resultado de la instanciación, se comporta como un paquete escrito directamente de la forma normal. La cláusula **use** nos permite referirnos directamente tanto a PUSH como a POP. Si hiciéramos una instanciación posterior

```
package ANOTHER_STACK is new STACK(50, INTEGER);
use ANOTHER_STACK;
```

entonces PUSH y POP son sobrecargas que pueden ser distinguidas por el tipo entregado por el contexto. Por supuesto, si ANOTHER_STACK también fuera declarado con el parámetro genérico real REAL, entonces deberíamos usar notación punto para distinguir las instancias PUSH y POP a pesar de las cláusulas **use**.

Tanto las unidades genéricas y las instanciaciones pueden ser unidades de biblioteca. De este modo, habiendo puesto el paquete genérico STACK en la biblioteca de programas se podría realizar una instanciación y compilarla separadamente..

```
with STACK;
package BOOLEAN_STACK is new STACK(200, BOOLEAN);
```

Si agregáramos una excepción de nombre ERROR al paquete, de tal modo que la declaración del paquete genérico fuese:

```
generic
        MAX: POSITIVE;
        type ITEM is private;
package STACK is
        ERROR: exception;
        procedure PUSH(X: ITEM);
        function POP return ITEM;
end STACK;
```

entonces cada instanciación debería dar origen a una excepción distinta y debido a que las excepciones no pueden ser sobrecargadas naturalmente tendríamos que usar la notación punto para distinguirlos.

Podríamos, por supuesto, hacer la excepción ERROR común a todas las instanciaciones definiéndola como global para todo el paquete genérico. Esta y el paquete genérico podrían quizá ser declarados dentro de otro paquete.

```
package ALL_STACK is
        ERROR: exception;
        generic
                MAX: POSITIVE;
                type ITEM is private;
        package  STACK is
                procedure PUSH(X: ITEM);
                function POP return ITEM;
        end STACK;
end ALL_STACKS;

package body ALL_STACK is
        package body STACK is
                ....
        end STACK;
end ALL_STACK;
```

Esto ilustra la ligación de los identificadores globales con las unidades genéricas. El significado de ERROR queda determinado en el lugar de la declaración genérica, independiente del significado que pudiese tener en el punto de instanciación.

Los ejemplos anteriores han ilustrado parámetros formales, los cuales eran tipos y también enteros. En efecto, los parámetros formales genéricos pueden ser cualquiera de los parámetros aplicables a subprogramas; pero también pueden ser tipos y subprogramas.

En el caso de los parámetros ya conocidos que también se aplican a subprogramas, estos pueden ser de modo **in** o **in out**, pero no **out**. Como con los subprogramas, **in** es tomado por omisión (como MAX en el ejemplo anterior).

Un parámetro genérico **in** actúa como una constante cuyo valor es entregado por el parámetro real correspondiente. Se permiten expresiones por omisión como en los parámetros de subprogramas; tal expresión es evaluada durante la instanciación si no se suministran los parámetros reales del mismo modo que en los subprogramas.

Un parámetro **in out**, actúa como una variable que renombra el parámetro real correspondiente. El parámetro real debe por tanto ser el nombre de una variable y su identificación ocurre en el punto de instanciación.

Nuestro último ejemplo en esta sección ilustra el anidamiento de genéricos. El siguiente procedimiento genérico realiza un intercambio cíclico de tres valores y está escrito en términos del procedimiento genérico EXCHANGE.

```
generic
        type THING is private;
procedure CAB(A, B, C: in out THING);

procedure CAB(A, B, C: in out THING) is
        procedure SWAP is new EXCHANGE(ITEM => THING);
begin
        SWAP(A, B);
        SWAP(A, C);
end CAB;
```

Aunque el anidamiento está permitido, este no debe ser recursivo.

Ejercicio.

1. Escriba la declaración de un paquete genérico que implemente el tipo abstracto de datos PILA (es decir, que se puedan definir variables de tipo PILA) de forma tal que se pueda variar el tamaño de la pila y el tipo que se pueda almacenar. Si se instanciasen dos paquetes:PILA_REALES y PILA_ENTEROS. ¿Qué problema habría al declarar, por ejemplo X:PILA? ¿Cómo se solucionaría el problema? ¿Habría problemas al usar directamente POP y PUSH?

2. Escriba un paquete genérico que permita definir variables de tipo ARREGLO a las cuales se le puede indicar el largo y el tipo de sus elementos. Sobre objetos tipo ARREGLO se pueden realizar las siguientes acciones: colocar (colocar un valor en una cierta posición), ordenar, invertir, primero (entrega el primer elemento) y último (entrega el último elemento).

11.2. Subprogramas como parámetros

Los parámetros genéricos también pueden ser subprogramas. En algunos lenguajes, como Algol y Pascal, los parámetros de subprogramas pueden a su vez ser subprogramas. Esta facilidad es útil para

aplicaciones matemáticas como la integración. En Ada, los subprogramas sólo pueden ser parámetros de unidades genéricas de modo que para estas aplicaciones se usa el mecanismo genérico.

Podríamos tener una función genérica

```
generic
        with function F(X: REAL) return REAL;
function INTEGRATE (A, B: REAL) return REAL;
```

la cual evalúa

$$\int_a^b f(x)dx$$

para integrar una función en particular debemos instanciar INTEGRATE con nuestra función como un parámetro genérico real. Así, supongamos que necesitamos integrar la función

$e^t \sin t$ entre los límites 0 y P

entonces escribiríamos

```
function G(T: REAL) return REAL is
begin
        return EXP(T)*SIN(T);
end;

function INTEGRATE_G is new INTEGRATE(G);
```

y nuestro problema queda resuelto mediante la expresión

```
INTEGRATE_G(0.0, P)
```

Nótese que un parámetro subprograma formal es como una declaración normal de subprograma precedida por **with**. (La palabra **with** al inicio es necesaria para evitar una ambigüedad sintáctica y no posee otro propósito.) La correspondencia entre subprogramas formales y reales es tal que el subprograma formal actúa sólo como un nuevo nombre para el subprograma real.

Ejercicio.

Dada la función

```
generic
        with function F(X: REAL) return REAL;
function SOLVE return REAL;
```

que encuentra una raíz de la ecuación f(x) = 0, muestre como encontraría la raíz de la ecuación

$e^x + x = 7$

¿Cómo haría que el tipo del parámetro de la función F también pudiese definirse durante la instanciación?

12. Tareas

El último tema principal a ser introducido es tareas. Esto ha sido dejado para el final, no porque no sea importante, pero si porque, aparte de la interacción con excepciones, es una parte bastante independiente dentro del lenguaje.

12.1 Paralelismo

Hasta ahora sólo hemos considerado programas secuenciales en los cuales las instrucciones son ejecutadas en orden. En muchas aplicaciones es conveniente escribir un programa con varias actividades paralelas las cuales interactúan como se requiera. Esto es particularmente cierto en programas que interactúan en tiempo real con procesos físicos en el mundo real.

En Ada, las actividades paralelas se describen por medio de tareas. En casos simples una tarea es léxicamente descrita por una forma muy similar a un paquete. Esto consiste en una especificación que describe la interfaz presentada a otras tareas y un cuerpo que describe el funcionamiento dinámico de las tareas.

```
task T is                    -- especificación
   .....
end T;

task body T is               -- cuerpo
   .....
end T;
```

En algunos casos una tarea no presenta interfaz a otras tareas en cuyo caso la especificación se reduce sólo a

```
task T;
```

Como un ejemplo simple de paralelismo considere una familia que va a comprar ingredientes para una comida. Suponga que necesitan carne, ensalada y vino; y la compra de esos artículos puede ser hecha al llamar a los procedimientos BUY_MEAT, BUY_SALAD y BUY_WINE respectivamente. El proceso completo podría ser representado por

```
procedure SHOPPING is
begin
    BUY_MEAT;
    BUY_SALAD;
    BUY_WINE;
end;
```

Sin embargo esta solución corresponde a la familia que compra cada artículo en secuencia. Sería más eficiente repartirse el trabajo de modo que, por ejemplo, la madre compra la carne, los niños compran la ensalada y el padre compra el vino. Y se ponen de acuerdo para encontrarse quizás en el estacionamiento. Esta solución paralela puede representarse por

```
procedure SHOPPING is
        task GET_SALAD;

        task body GET_SALAD is
        begin
                BUY_SALAD;
        end GET_SALAD;

        task GET_WINE;

        task body GET_WINE is
        begin
                BUY_WINE;
        end GET_WINE;

begin
        BUY_MEAT;
end SHOPPING;
```

En esta formulación, la mamá se representa como el proceso principal y llama a BUY_MEAT directamente desde el procedimiento SHOPPING. Los niños y el papá se consideran como procesos subordinados y ejecutan las tareas localmente declaradas GET_SALAD and GET_WINE las cuales respectivamente llaman a los procedimientos BUY_SALAD y BUY_WINE.

El ejemplo ilustra la declaración, activación y terminación de tareas. Una tarea es un componente de un programa como un paquete y se declara en forma similar dentro de un subprograma, bloque, paquete o en otro cuerpo de tarea. Una especificación de tarea puede también declararse en una especificación del paquete en cuyo caso el cuerpo de tarea debe declararse en el paquete del cuerpo correspondiente. Sin embargo, una especificación de tarea no puede declararse en la especificación de otra tarea, sino sólo en el cuerpo.

La activación de una tarea es automática. En el ejemplo anterior las tareas locales se activan cuando la unidad de los padre alcanza el **begin** que sigue a la declaración de las tareas.

Tal tarea terminará cuando alcance su final **end.** Así la tarea GET_SALAD llama al procedimiento BUY_SALAD y luego termina rápidamente.

Una tarea declarada en la parte declarativa de un subprograma, bloque o cuerpo de tarea se dice que depende de esa unidad. Una regla importante es que una unidad no puede ser dejada hasta que todas las tareas dependientes hayan terminado. Esta regla asegura que los objetos declarados en la unidad, y por esto mismo potencialmente visible a las tareas locales, no pueden desaparecer mientras exista una tarea que pueda accesarlos.

Es importante darse cuenta que se considera que el programa principal es llamado por una hipotética "tarea principal". Ahora podemos indicar la secuencia de acciones cuando esta tarea principal llama al procedimiento SHOPPING. Primero se declaran las tareas GET_SALAD y GET_WINE y luego cuando la tarea principal alcanza el **begin** estas tareas dependientes son activadas en paralelo con la tarea principal. Las tareas dependientes llaman a sus respectivos procedimientos y finalizan. Mientras, la tarea principal llama a BUY_MEAT y luego alcanza el **end** de SHOPPING. Luego, la tarea principal espera hasta que las tareas dependientes hayan terminado si es que todavía no lo han hecho. Esto corresponde a la madre esperando al padre y a los niños que vuelven con sus compras.

En el caso general la terminación ocurre en dos etapas. Decimos que una unidad se ***completa*** cuando alcanza su **end** final**.** Y se le considerará ***terminada*** cuando todas las tareas dependientes, si hay algunas, también estén terminadas. Por supuesto, si una unidad no tiene tareas dependientes entonces se completa y termina al mismo tiempo.

Ejercicio

Reescriba el procedimiento SHOPPING de modo que contenga 3 tareas y así revele la simetría natural de la situación. (Es decir, que las tres acciones tienen la misma jerarquía.)

12.2 El rendezvous

En el ejemplo SHOPPING las tareas no interactuaron unas con otras una vez que han sido activadas, excepto por el hecho que la unidad padre tiene que esperar a que todas terminen. Sin embargo, lo que ocurre generalemente es que las tareas interactúan y se coordinan para la consecución de un fin común. En Ada esto se ejecuta por un mecanismo conocido como un **rendezvous**. Esto es similar a la situación humana donde dos personas se encuentran, ejecutan una transacción y luego continúan independientemente.

Un rendezvous entre dos tareas ocurre como consecuencia de la llamada de una tarea a una entrada (**entry**) declarada en otra tarea. Una entrada se declara de un manera similar a como se declara un procedimiento en la especificación de un paquete.

```
task T is
        entry E( . . . );
end;
```

Una entrada puede tener parámetros **in, out** e **in out** al igual que un procedimiento. Sin embargo no puede entregar un resultado como una función. Una entrada es llamada de una manera similar a un procedimiento

```
T.E( . . .);
```

Un nombre de tarea no puede aparecer en una cláusula **use** y por este motivo se requiere la notación punto para llamar la entrada desde el exterior de la tarea. Por supuesto, una tarea local podría llamar a una entrada de sus padres directamente - se aplican las reglas de visibilidad y alcance usuales.

Las instrucciones que se ejecutan durante un rendezvous por cada entrada son descritas por instrucciones de aceptación (**accept**) correspondientes en el cuerpo de la tarea. Una instrucción de aceptación generalmente toma la forma

```
accept E( . . . ) do
        - - secuencia de instrucciones
```

```
end E;
```

Los parámetros formales de la entrada **E** son repetidos del mismo modo que el cuerpo de procedimiento repite los parámetros formales de su declaración. El **end** opcionalmente es seguido por el nombre de la entrada. Una diferencia importante es que el cuerpo de las instrucciones **accept** es sólo una secuencia de instrucciones. Cualquiera declaración local o manejo de excepción deben ser suministradas escribiendo un bloque local.

La diferencia más importante entre una llamada de entrada y una llamada de procedimiento es que en el caso de un procedimiento, la tarea que llama al procedimiento inmediatamente también ejecuta el cuerpo de procedimiento mientras en el caso de una entrada, una tarea llama a la entrada pero la instrucción de aceptación correspondiente es ejecutada por la tarea que posee la entrada. Además, la instrucción de aceptación no puede ser ejecutada hasta que una tarea llama a la entrada y la tarea que posee la entrada "llega hasta" la instrucción de aceptación. Naturalmente uno de estos acontecimientos ocurrirá primero y la tarea que esta involucrada (la que llama) queda suspendida hasta que otra (la poseedora de la entrada) llega a su instrucción (**accept**) correspondiente. Cuando esto ocurre se ejecuta la secuencia de instrucciones de la instrucción de aceptación. A esta interacción se le llama ***rendezvous***. El ***rendezvous*** se completa cuando se alcanza el fin de la instrucción de aceptación, luego ambas tareas proceden en forma independiente.

Podemos elaborar nuestro ejemplo de compra dando a la tarea GET_SALAD dos entradas, una para la madre para que de dinero a los niños para ensalada y una para recoger su ensalada después. Hacemos lo mismo para GET_WINE .

También podemos reemplazar los procedimientos BUY_SALAD, BUY_WINE y BUY_MEAT por funciones que toman dinero como un parámetro y retorna el ingrediente apropiado. Nuestro procedimiento compra ahora sería

```
procedure SHOPPING is
    task GET_SALAD is
        entry PAY(M: in MONEY);
        entry COLLECT(S: out SALAD);
    end GET_SALAD;

    task body GET_SALAD is
        CASH: MONEY;
        FOOD:SALAD;
    begin
        accept PAY(M: in MONEY) do
            CASH:=M;
        end PAY;

        FOOD:=BUY_SALAD(CASH);

        accept COLLECT(S: out SALAD) do
            S:=FOOD;
        end COLLECT;
    end GET_SALAD;

    - - GET_WINE en forma similar

    begin
        GET_SALAD.PAY (50);
        GET_WINE.PAY (100);
        MM:=BUY_MEAT (200);
        GET_SALAD.COLLECT (SS);
        GET_WINE.COLLECT (WW);
    end;
```

El resultado final es que varios ingredientes terminen en las variables MM, SS, y WW cuyas declaraciones son dejadas a la imaginación.

Conviene analizar el comportamiento lógico del ejemplo presentado. Tan pronto como las tareas GET_SALAD y GET_WINE se activan encuentran instrucciones **accept** y esperan hasta que la tarea principal llame las entradas PAY en cada una de ellas. Después de llamar a la función BUY_MEAT, la tarea principal llama a las entradas COLLECT. Curiosamente, la madre no esta habilitada para entregar el dinero para el vino hasta después que halla entregado el de la ensalada. Una situación análoga se presenta con la recolección.

Como un ejemplo más abstracto consideremos el problema de crear una tarea que actúe como un simple buffer entre una o más tareas que producen itemes y una o más tareas que los consumen. Nuestra tarea intermedia puede mantener solo un ítem.

```
task BUFFERING is
```

```
        entry PUT(X: in ITEM);
        entry GET(X: out ITEM);
end;

task body BUFFERING is
        V: ITEM;
begin
        loop
                accept PUT(X: in ITEM) do
                        V:=X;
                end PUT;
                accept GET(X: out ITEM) do
                        X:=V;
                end GET;
        end loop;
end BUFFERING;
```

Entonces otras tareas pueden colocar o adquirir itemes llamando

```
BUFFERING.PUT(. . .);
BUFFERING.GET(. . .);
```

El almacenamiento intermedio para el ítem es la variable V. El cuerpo de tarea es una iteración infinita que contiene una instrucción de aceptación para PUT seguido por GET. De este modo la tarea acepta en forma alternada llamadas de PUT y GET las cuales llenan y vacían la variable V.

Varias tareas diferentes pueden llamar a PUT y GET y consecuentemente tendrán que ser enfiladas. Cada entrada (**entry**) tiene asociada una fila de tareas a la espera de llamar a la entrada - esta fila es procesada en modo FIFO y puede, por supuesto, estar vacía en un momento dado. La cantidad de las tareas en la fila de la entrada E es dado por E'COUNT pero este atributo puede ser usado sólo dentro del cuerpo de la tarea que posea la entrada.

Un entrada puede tener varias instrucciones de aceptación (generalmente sólo una). Cada ejecución de una instrucción de aceptación remueve una tarea a la fila.

Note la asimetría intrínseca del rendezvous. La tarea que se llama debe nombrar a la tarea llamada pero no viceversa. Además, muchas tareas pueden llamar una entrada y ser puestas en fila pero una tarea puede estar sólo en una fila a la vez .

Una entrada puede no tener ningún parámetro, tal como

```
entry SIGNAL;
```

y podría luego ser llamada por

```
T.SIGNAL;
```

Una instrucción de aceptación no necesita tener cuerpo como en

```
accept SIGNAL;
```

En tal caso el propósito de la llamada es simplemente efectuar una sincronización y no pasar información.

No hay restricciones sobre las instrucciones en una instrucción **accept**. Ellas pueden incluir llamadas de entradas, llamadas de subprogramas, bloques e incluso otras instrucciones de aceptación (pero no para la misma entrada). Por otro lado, una instrucción de aceptación no puede aparecer en el cuerpo de un subprograma, sino que debe estar en la secuencia de instrucciones de la tarea, aunque podría estar en un bloque o dentro de otra instrucción de aceptación. La ejecución de una instrucción de retorno (**return**) en una instrucción de aceptación corresponde a alcanzar el final y por lo tanto termina el rendezvous.

Una tarea puede llamar a una de sus propias entradas pero, por supuesto, esto provocará un deadlock. Esto puede parecer disparatado pero los lenguajes de programación permiten gran cantidad de cosas "tontas" tales como una iteración infinita, etc.

Ejercicios

1. Escriba el cuerpo de una tarea cuya especificación es

```
task BUILD_COMPLEX is
        entry PUT_RL(X: REAL);
        entry PUT_IM(X: REAL);
        entry GET_COMP(X: out COMPLEX);
end;
```

y que alternadamente crea un número complejo por medio de las lamadas a PUT_RL y PUT_IM y luego envíe el resultado en una llamada de GET_COMP.

2. Escriba el cuerpo de una tarea cuya especificación sea

```
task CHAR_TO_LINE is
        entry PUT(C: in CHARACTER);
        entry GET(L: out LINE);
end;
```

donde

```
type LINE is array (1 . . 80) of CHARACTER;
```

La tarea actúa como un buffer que alternadamente construye una línea aceptando llamadas sucesivas de PUT y luego entrega una línea completa en una llamada de GET.

3. En INTERNET existe una página donde se puede jugar un juego de apuestas colectivo que consiste en que se le hacen preguntas a un personaje y se debe determinar su identidad.

Existe un personaje enmascarado al cual se le pueden formular preguntas, a las cuales puedo o no responder (por ejemplo, no responderá si se le pregunta su nombre). Para participar, una persona debe inscribirse: hace su apuesta y recibe una clave que le permite participar en el juego. Tiene derecho a una o más preguntas (no necesariamente consecutivas) y por cada una de ellas puede formular una identificación del personaje. Si acierta, recibe 20 veces su apuesta. Las personas que se han inscrito y no alcanzan a participar pierden su apuesta.

El programa que controla el juego tiene dos tareas paralelas: una que controla la inscripción y otra que controla el juego propiamente tal. Cuando alguien gana, las inscripciones se suspenden.