ESTRUCTURA DE UN PROGRAMA.

Un programa consta de dos partes: la parte declarativa, que describe los datos que serán usados en el programa, y la parte de instrucciones. La estructura de un programa en ADA es esta:

procedure nombre_programa is

parte_declarativa

begin

parte_desarrollo

end nombre_programa

VARIABLES Y TIPOS BASICOS.

Nombre de las variables.

El nombre de las variables puede ser el que el programador quiera, siempre que no sea ninguno de los nombres restrictivos que tiene ADA.

Tipo de variables.

El tipo de una variable define el repertorio de valores que una variable puede tomar. El tipo también indica las operacines que se podrán aplicar sobre una variable. El tipo de una variable también nos dice el espacio de memoria que se utilizará para cada variable.

Ada tiene algunos tipos predefinidos. Los más importantes son los booleanos, caracteres y enteros. Las booleanas pueden tomar dos valores, true o false. Los caracteres son todos los símbolos que se pueden dar con las teclas. Los enteros pueden tomar cualquier valor del tipo número entero.

Valor de las variables.

Una variable siempre tiene un valor aunque el programador no le halla dado ninguno. Ese valor es impredecible.

Declaración de las variables.

Cuando un programa necesite alguna variable, ésta será declarada en la parte declaraiva del programa. La declaración significa dar un nombre a una variable y un tipo.

SECUENCIAS, CONDICIONALES Y LOOPS.

Composición secuencial: significa que se llevan a cabo las instrucciones una detrás de la otra.

Composición condiconal: son operaciones que sólo se ejecutan si se da una condición dad por el programador.

Loops: son operaciones que se repiten hasta que se llega a una condición para parar de repetirlas. Estas operaciones pueden no empezar a ejecutarse si por ejemplo cuando se llega al principio del loop ya se ha

cumplido la condición necesaria.

INPUTS Y OUTPUTS D LOS TIPOS BASICOS.

get(c): espera para recibir un carácter llegado desde el teclado. El carácter entonces es almacenado en la variable especificada entre los paréntesis, en este caso c.

put(c): escribe el valor de la variable en la pantalla del ordenador.

Estas funciones vienen dadas en la librería ada.text_io.

También suele ser necesaria la librería ada.integer_text_io.

COMPILACIÓN.

Es el traslado del lenguaje utilizado para hacer los programas, al lenguaje máquina.

SUBPROGRAMAS.

Son programas que están dentro de los programas principales y que sirven para llevar a cabo funciones dentro del programa principal. Los problemas de cada programa son rotos en pequeños subproblemas más fáciles de resolver por los llamados subprogramas.

DECLARACION E INVOCACION DE LOS SUBPROGRAMAS.

Los subprogramas deben ser declarados antes de ser usados. La declaración de un subprograma incluye su nombre y su cuerpo. Una vez que un subprograma es declarado, puede ser usado tantas veces como sea necesario.

VISIBILIDAD.

La parte declarativa de un subprograma contiene los parámetros que serán usados para resolver el subprograma y esos parámetros son irrelevantes para los demás procedures. Un subprograma también puede contener dentro otros subprogramas. Los parámetros locales pueden tener el mismo nombre que los demás y esto no supone ningún peligro de error dentro del programa.

COMPILACION SEPARADA(SEPARATES).

Los subprogramas también pueden ser escritos de forma separada en archivos diferentes del programa principal. La compilación separada tiene muchas ventajas. Por una parte los ficheros son mucho más pequeños y fáciles de editar y cambiar. Por otra parte cuando se hace algún cambio, sólo tienen que ser compiladas las partes que se han cambiado y no todos los programas, y por tanto el proceso de compilación es mucho más rápido.

El fichero con el *nombre.adb* debe ser compilado primero(que es el programa principal). El programa principal puede ser compilado antes de hacer el subprograma porque el subprograma hace referencia a parámetros del programa principal. Por la misma razón, si el programa principal es actualizado y recompilado, el subproblema también podrá ser compilado otra vez si no se han introducido cambios desde la última compilación. Después de corregir un fallo o problema en un subprograma, podemos compilar sólo el subprograma sin necesidad de volver a compilar todo el programa principal.

ARRAYS(colección de...)

Un array en un conjunto de componentes del mismo tipo. Cada componente puede ser distinguida individualmente por el significado de un índice, el cual puede ser calculado al mismo tiempo.

A fin de definir cada tipo de array, la extensión de los índices y el tipo de componentes deben ser dados. La extensión del índice implica, en particular el número de componentes del array. La declaración de un array será como:

type nombre is array(mín_valor..max_valor) of tipo de los componentes;

Cuando el tipo ya esté definido, podremos declarar tantas variables como sean necesarias para cada tipo. La manera de declarar ua variable de tipo array sería pr ejemplo:

a: nombre;

Esta declaración indica que *a* es una variable de tipo array. Por tanto, el nombre *a* se refiere a la palabra entera, mientras que los componentes individuales son referidos añadiendo un índice a la *a* , entre paréntesis: a(i).

El índice es una expresión que puede ser evaluada al tiempo. Por tanto, si una operación se ha de aplicar a diferentes componentes del array, el tratamiento puede ser incluido en un *loop* en el que el índice es modificado para cada vez que se realice el loop. Si durante la ejecución del programa el índice evalúa un valor que no está especificado entre la extensión especificada, ocurrirá un error.

Nosotros podemos definir un rango medio para todas las palabras, pero como es lógico, no todas las palabras tendrán la misma longitud, por lo que definiremos una longitud suficiente para mantener todas las palabras posibles. Además, seremos capaces de distinguir, entre los componentes, cuáles tienen mucha información y cuáles no. Una posibilidad es usar una marca especial como un espacio en blanco al final de las palabras significativas.

RECORDS.

Los records son un conjunto de componentes de diferentes tipos. Para definir cada tipo de record, los diferentes componentes han de ser numerados. Para cada componente, el nombre y el tipo han de ser dados. Así que la declaración de un record sera:

type nombre is

record

nombre_comp1: tipo;

nombre_compn: tipo;

end record;

Una vez que el tipo está definido, podemos declarar tantas variables del mismo tipo como queramos, como hacemos con tipos predefinidos:

a: nombre;

Esta declaración indica que *a* es una variable de tipo record. Por tanto, el nombre *a* denota el conjunto completo de componentes, mientras que las componentes individuales, se escriben añadiendo el nombre del

componente a la *a*, separados con un punto: a.nombre_componente.

Volviendo atrás, podemos decidir no almacenar una marca especial para distinguir todos los componentes significativos y sin usar de nuestro array, pero para mantener la longitud de las palabras almacenadas explícitamente.

Por tanto, la palabra es representada por ambos conjuntos(array) de caracteres y el entero que contiene su longitud. Mantenerlos en diferentes variables es algo confuso. Mucho más apropiado es definir un tipo especial que esté compuesto por el conjunto más un componente entero. Para el caso particular de las palabras la declaración sería:

type contenido_palabra is array(1..19) of character;

type palabra is

record

contenido: contenido_palabra;

longitud: entero;

end record;

DECLARACIONES LOCALES.

Una variable por ejemplo i, que sólo se usa en un subprograma, puede ser declarada en el programa principal. Sin embargo, es inapropiado hacerlo porque i solo se usa localmente en el subprograma. Su valor es irrelevante fuera de ese procedure. Los subprogramas también tienen parte declarativa y por tanto deben declararse las variables que se van a usar en ese subprograma y siempre es bueno que cada subprograma declare las variables que vaya a usar dentro del mismo y en ningún otro sitio para evitar errores.

Además, de esta manera, se pueden coger más de una variable local con el mismo nombre ya que esto no afectará al correcto funcionamiento del programa.

PARÁMETROS Y FUNCIONES.

Los parámetros nos indican el tipo de variable que vamos a utilizar en el sentido, por ejemplo, de si es de lectura o de escritura por ejemplo. La forma de declarar parámetros sería:

procedure p(x:in out tx; y: in ty; z: out tz);

La dependencia es descrita como un grupo de nombres, juntas con sus tipos y modos. Una declaración como la anterior significa que el procedure tiene tres parámetros, que podrán ser llamadas con tres parámetros: p(a,b,c); o p(r,s,t);. Los nombres que aparecen en el encabezamiento del procedure son normalmente llamados *parámetros formales*, porque no hacen referencia a muchas variables o constantes

particulares. Los nombres que aparecen en las llamadas (a,b,c) y (r,s,t) se llaman *parámetros efectivos*, porque indican los objetos efectivos sobre los cuales el procedimiento actuará: cuando se invoca el procedure con los parámetros(a,b,c), las operaciones que están hechas en el cuerpo del procedure para *x*

serán hechas efectivamente para *a*, mientras que si es invocado con los parámetros(r,s,t) las operaciones se harán para *r*. Obviamente, los parámetros deberán ser del mismo tipo, de otra manera la misma operación se

podría aplicar a todos ellos. Por esta razón, el tipo de los parámetros es especificado en el encabezamiento del procedure y el compilador no admitirá ninguna llamada con parámetros efectivos cuyos tipos no sean iguales que los de los parámetros formales.

Además, el encabezamiento de los procedures también especifican el modo de los parámetros. Si el modo es **in,** el procedure no tiene permiso para modificar ese parámetro, sólo para consultarlo. Por tanto, ese parámetro formal será considerado como una constante y cualquier asignamiento que se le de será considerado como un error para el compilador. Si el parámetro formal es **out**, el procedure puede modificar su valor. Finalmente, si el modo es **in out**, el procedure puede consultar y modoficar ese parámetro, así que el resultado del procedure, puede depender del valor que le corresponda al parámetro cuando el procedure es llamado.

Los parámetros del modo **out** o **in out**, deben ser necesariamente una variable. En cambio, los parámetros del modo **in** pueden ser expresiones que tengan como resultado un valor del tipo esperado. En particular, una variable es una expresión y una constante es también una expresión.

VISIVILIDAD.

La estructura de los procedures cuando están en un programa, es exactamente la misma. Tienen una parte declarativa y una parte de instrucciones.

El principio de la abstracción implica que el algoritmo que usa un procedure debe ser independiente de la manera de invocar dicho procedure. El procedure debe relatar lo que hace, no cómo lo hace. Por tanto, el algoritmo que usa un procedure debería permanecer intacto si el procedure es cambiado por otro que hace exactamente lo mismo.

Esta independencia sólo puede permanecer si cualquier variable, tipo, subprograma, que ha aparecido en el análisis de un subproblema es declarado localmente en el procedure que resuelve ese subproblema.

Los nombres de las cosas declaradas en una misma parte declarativa deben ser diferentes porque cada variable con un nombre sólo puede desempeñar una función y no más de una. Pero si hacemos un procedure independiente, nos podemos encontrar que al juntarlo con otro procedure tengamos variables con el mismo nombre en los dos procedures. Este principio levanta la cuestión de visivilidad.

Si una entidad local tiene el mismo nombre que una global, el significado del nombre de la local es el mismo que en la declaración local.

Cada variable declarada debe ser utilizada dentro del programa en el que se ha declarado. Si no es así, deberemos indicar, entre paréntesis, que la variable que vamos a utilizar corresponde a la parte declarativa a la que hallamos hecho referencia.

VARIABLES GLOBALES CONSIDERADAS DAÑINAS.

Los procedures que actúan sobre variables globales son mucho más difíciles de usar y entender que los que cuyos efectos dependen sólo de datos que son pasados como parámetros. Usar un procedure que opera con variables globales es incómodo porque el programador que utiliza ese procedure tiene que analizar y mantener en mente cuáles de las variables pueden ser afectadas por cada invocación del procedure. Analizar un procedure que opera con variables globales es también pesado porque es forzado a mantener la mente para ser capaz de entender cómo proceder y por qué.

Además los procedures que utilizan variables globales pueden ser utilizados de menos maneras porque son mucho más dependientes de su contenido y su uso compromete el mantenimiento de los programas. Si por una razón de mantenimiento, el nombre de una variable en un programa que usa cada procedure es

modificado, puede pasar que el procedure no trabaje mucho.

Además con estas variables, algunos de los errores que de otra manera serían detectados por el compilador podrían no serlo. Un programador que está resolviendo un subprograma puede usar un índice y darle un nombre para los índices, que nosotros denominamos *i.* Si olvida declararlo y el procedure aparece en un campo dónde las variables globales no son visibles, el error se noticiará y será corregido con un pequeño esfuerzo. Pero si el procedure aparece en un campo donde otro entero llamado *i* exista, aparecerá un error de no compilación. En cambio, la variable global se usará para propósitos globales y su valor puede ser inesperadamente cambiado por el procedure invocado y el programa arrancará dando malos resultados.

Una variable puede ser visible siendo global o como un parámetro y puede causar indeseados efectos de alias, que significan que la misma variable es conocida por más de un nombre.

TIPOS DE ENUMERACIÓN.

Es preferible tener un tipo cuyo valor fuese, por ejemplo, *cifrar* y *descifrar* y declarar el orden de la variable como siendo de ese tipo. Los modernos lenguajes de programación permiten al programador definir cada tipo de *tipo* −llamados tipos de enumeración− y ADA también lo hace.

Declarar un tipo de enumeración en ADA se hace por una declaración donde un nombre se da a cada uno de los diferentes valores que maquillarán el tipo. Por ejemplo:

type command_type is(cipher, decipher);

Cada declaración define tres nombres: primero el tipo_comando, que está para un nombre *tipo*. Segundo, los nombres cipher y decipher, que están para constantes de cada tipo de esos. Las variables pueden entonces ser declaradas de cada tipo de la manera normal:

Una variable contendrá sólo uno de los dos valores permitidos:

com1:=cipher;

com2:=decipher;

com1:=com2;

Estos son correctos, los siguientes son incorrectos:

com1:='c';

com2:=true;

com1:=0;

com2:=c; −−asumiendo que c es una variable de tipo carácter.

c:=com2;

Cuando diferentes acciones han de ser hechas acuerdo con el valor dado en una variable del tipo enumeración, una estructura específica de control, el caso dado, permite las diferentes acciones para ser declaradas simétricamente. El caso dado tendría la forma:

case com1 is

when cipher => acciones_para_cipher;

when decipher => acciones_para_decipher;

end case;

Si un tipo de enumeración tiene varios valores para los cuales la misma acción ha de ser dada, una alternativa adicional puede ser añadida al final:

case com1 is

...

when others => acciones_para_otros;

end case;

Las acciones para los otros serán ejecutadas si el valor de com1 no es de las alternativas anteriores.

PACKAGES.

Un package en una construcción para declaraciones relatadas en grupos cerradamente. Esencialmente, los tipos y sus procedures y functions relatados. Además, la construcción package nos suministra los significados de los detalles que son irrelevantes para otros componentes del programa que hacen uso de las entidades declaradas en el package.

Un package se compone de dos partes: su parte de especificación y su cuerpo(*body).* En principio, la parte de especificación es para enseñar las facilidades que el package administra a otros componentes del programa, y el body es la parte donde se escribe la implementación de cada una de esas facilidades. Sin embargo, compilaciones reducidas está está forzada la inclusión de algunos aspectos de implementación en la parte de especificación, así que el compilador se puede enterar de lo suyo para generar el código apropiado. Esos aspectos, que el programador quiere para ser ocultados se encuentran en una parte de la parte de especificación que se llama *private* . La parte de especificación y el body pueden ser compilados como unidades independientes del programa, como el programa principal, o pueden aparecer en alguna parte declarativa de otra unidad de programa. En este último caso, las partes de especificación y el body deben aparecer en la misma parte declarativa, aunque el body pueda ser compilado separadamente. La estructura de un package es sí:

package nombre_package is

−−cosas dadas por el package para uso externo.

private

−−aspectos de implementación de las cosas dadas que son requeridos por el compilador para computar los tamaños de los tipos. Ellos no son visibles para los componentes del programa que hacen uso del package.

end nombre_package.

Un uso principal de los packages es para definir tipos abstractos de datos.

Lo que los packages permiten hacer a los programadores es definir sus convenientes tipos con las mismas propiedades que los tipos predefinidos que tenemos. Esta es una definición de un tipo privado:

package d_t is

type t is private;

−−cabeceras de los procedures y functions relatados para el tipo.

private

type t is ...; −−representación del tipo.

end d_t;

Esta construcción significa que el tipo **t** puede ser usado por todos los componentes donde el package d_t está visible. Pero cuando un tipo es definido como privado dentro del package sólo es visible dentro de ese package y por lo tanto no puede ser usado fuera de él. Así que, todas las operaciones que utilicen el tipo **t** deberan estar compuestas usando las operaciones ofrecidas por el package como primitivas. Una consecuencia directa es que si se encuentra una mejor representación para el tipo, el código afectado son sólo las operaciones declaradas en ese mismo package.

La implementación de esas operaciones tiene que ser puesta en el package body, aunque éste pueda ser compilado por separado:

package body d_t is

−−cuerpos completos de los procedures que tienen la cabecera en la parte de especificación, además de algunas declaraciones auxiliares que pueden ser requeridas.

end d_t;

ORDEN DE COMPILACIÓN.

El orden de la compilación es el siguiente:

Primero se compila el programa principal, luego los package y más tarde, si los hemos hecho, se compilan los separates correspondientes a cada parte de los package.

PRAGMA INLINE.

La definición de los lenguajes incluyen cláusulas llamadas *pragmas*, que fueron introducidas en el lenguaje como directrices para el compilador. Estas no cambian el significado del programa. Un pragma importante es el pragma **inline** . Si el programador incluye una cláusula en cada parte de esfecificación del package como

package ... is

type ... is private;

procedure inicializar...;

...

```
pragma inline(inicializar);

private

--lo mismo que arriba.

end ...;
```

El programa hará lo mismo que antes. Sin embargo, el código generado para la llamada de *inicializar* no será una llamada de procedure efectiva sino un repuesto del código. Esto es igual de eficiente como si hubiéramos hecho la llamada del procedure directamente.

ABSTRACCIÓN DE UNA SECUENCIA DE PALABRAS DE UNA SECUENCIA DE CARACTERES.

De acuerdo con el problema nosotros tenemos una frase acabada en punto y las palabras pueden estar separadas por uno o más espacios en blanco y puede haber ninguno o algún espacio en blanco antes de la primera palabra y entre la última palabra y el punto final.

Este procedure para coger una palabra es fácil en principio pero no lo es cuando queremos garantías de que irá bien cuando sea llamado. Para que vaya bien, después de que halla cogido una palabra, el carácter debe ser puesto al principio de la siguiente palabra, si la hay, y sino sebe ser puesto en el punto. Inicialmente debe ser puesto en el primer carácter de la primera palabra.

La construcción del package nos permite esconder cada carácter dentro del body, así que finalmente desaparecen del nivel erróneo de abstracción. La construcción del package también permite al programdor introducir un bloque de operaciones para dar valores iniciales a las variables declaradas en el cuerpo del package. En nuestro caso, para poner el carácter al principio de la primera palabra. Esto nos lleva a la siguiente parte de implementación:

```
With ada.text_io; use ada.text_io;

package body d_palabra is

c:character;

procedure copiar(s: in palabra; t: out palabra) is separate;

function igual( p1, p2: in palabra) return boolean is separate;

procedure tomar(w: out palabra) is separate;

procedure put( w: in palabra); is separate;

function quedan_palabras return boolean is separate;

begin

get(c);

while c=' ` loop

get(c);
```

```
end loop;

end d_palabra;
```

Después vienen los siguientes procedues:

```
separate( d_palabra)

procedure tomar( w: out palabra) is

i: indice_c_palabra;

begin

i:=0;

while c/='.' and c/=' ` loop

i:= i+1; w(i):=c;

get(c);

end loop;

i:=i+1; w(i):=' `;

while c=' ` loop

get(c);

end loop;

end tomar;

separate( d_palabra)

function quedan_palabras return boolean is

begin

return c/='.'

end quedan_palabras;
```

UNIDADES GENÉRICAS.

Si nos damos cuenta muchas veces utilizamos procedures y algoritmos que son muy parecidos y que sólo cambian en las variables y poco más, Los tipos genéricos sirven para declarar una sola vez un tipo que vamos a utilizar muchas veces y así no lo tenemos que declarar cada vez. Un tipo típico de genérico son los tipor array.

FICHEROS DE TEXTO.

Un fichero de texto es una secuencia de caracteres que están almacenados en algunas tablas permanentes apoyados como un disco. Los caracteres en el fichero pueden ser leídos secuencialmente como hemos hecho hasta ahora con los caracteres procedentes del teclado, con la única diferencia que ahora el origen de los caracteres debe ser especificado. Esta especificación es archivada dándole al sistema el nombre del fichero que queremos que sea el origen para nuestras operaciones de obtención.

Una referencia específica es requerida por el fichero en cada operación de obtención. La operación de apertura del fichero, asocia el nombre del fichero a una referencia de este tipo. Cuando el programa no más corto necesita acceder al fichero, puede liberar la asociación entre la referencia interna y el fichero. Esto es llamado para cerrar el fichero.

Los programas también pueden almacenar datos en ficheros, y pueden hacerlo como hemos hecho hasta ahora para enseñar los resultados, con la única diferencia de que ahora la destinación de los caracteres debe ser especificada. Las operaciones *put* requerirán una referencia para la destinación del fichero como un parámetro adicional. La destinación del fichero puede ser cualquier fichero que ya exista para lo cual los caracteres escritos son añadidos al final o un fichero que está creado por el programa mismo.

Sin embargo, un fichero de texto como cualquier fichero secuencial, no puede ser sobreescrito mientras esté siendo leído, ni tampoco puede ser leído mientras esté siendo escrito.

Un fichero es abierto de algún modoesencialmente leído o escritolo cual es especificado cuando el fichero es abierto y mantiene lo mismo hasta que se cierra.

Los mecanismos para administrar los ficheros de texto es dado por el package de ADA ada.text_io .

IDENTIFICAR FICHEROS.

Por algunas razones, el tipo para designar ficheros es declarado como *limited private* en el package ada.text_io:

type tipo_fichero is limited private;

Para leer o escribir caracteres a un fichero, una variable del tipo anterior debe ser asociada con un fichero particular, que se hace a través de una llamada al procedure abrir, que se especifica como:

procedure abrir(fichero: in out tipo_fichero; modo: in modo_fichero; nombre: in string)

El modo indica si el fichero es proyectado para ser usado para leer o escribir. Los diferentes modos son declarados en el package ada.text_io como un tipo enumeración.

type modo_fichero is (in_fichero, out_fichero, fichero_ añadido);

Como los nombres indican, el modo in_fichero es para usar el fichero en modo lectura, el modo out_fichero es para reescribir el fichero− destruyendo sus anteriores contenidos− y el modo fichero_añadido, escribe en el fichero así como el anterior contenido en mantenido y escribiendo es hecho como añadido a los caracteres al final del fichero.

Cuando la operación en un fichero se ha completado, puede ser liberado con la operación cerrar:

procedure cerrar(fichero: in tipo_fichero);

LEYENDO Y ESCRIBIENDO.

Los caracteres pueden ser cogidos de un fichero que ha sido abierto con in_fichero con el significado de la operación get(tomar).

procedure tomar(fichero: in tipo_fichero; item: out character);

Los caracteres pueden ser añadidos al fichero que ha sido abierto con el out_fichero o fichero_añadido, con la operación put(poner);

procedure put(fichero: tipo_fichero; item: in character);

EL PACKAGE ESTANDARD *ADA.COMMAND_LINE*.

El package standard comand_line suministra el significado para obtener cada argumento desde la línea de comandos.

Éste proporciona dos funciones útiles:

function argument_count return natural;

function argument(numero: in positivo) return string;

Los tipos natural y positivo están declarados en el package estandard, siendo *natural* un subtipo de enteros desde el uno, y *positivo* un subtipo de enteros desde el uno. El tipo natural, se usa para contar el número de argumentos en el comando, que puede ser cero, mientras que el tipo positivo se usa para seleccionar el argumento particular que realmente queremos.

El nombre del argumento es dado como una cadena de caracteres por el argumento function desde el argumento número. La cadena resultante puede ser usada como argumento para otros procedures, en particular open(abrir) y put(poner).

NUEVOS DETALLES DEL LENGUAJE.

Los Strings, como cualquier variable de tipo longitud, son complejos y su completa introducción es detallada en el segundo volumen de este trabajo, donde pueden ser tratados como casos particulares de tipos inconstantes. Sin embargo, ciertos usos elementales de cadenas, como llamadas del tipo

put(mensaje);

o

open(f, in_fichero, nombre_fichero);

no requieren su entendimiento completo. Similarmente, para construir un nuevo nombre para uno previo extendiéndolo con un sufijo y pasando el resultado a un procedure que espera una cadena como un parámetro, la única característica requerida es el operador de encadenamiento −&−, que se usa como:

open(f, in_fichero, nombre_fichero & ext);

COMPARANDO PALABRAS.