TEMA 1: COMPONENTES REUTILIZABLES DEL SOFTWARE.

1.1 DISEÑO MODULAR Y COMPONENTES SOFTWARE.

El principal objetivo de lo que se conoce como Ingeniería del Software es producir software de calidad. El ideal que siempre persigue un informático es que sus programas sean rápidos fiables, fáciles de usar, legibles, modulares, estructurados, etc. Estas características describen dos tipos diferentes de cualidades software.

Por un lado estamos considerando propiedades como facilidad de uso y rapidez, que son detectadas por los usuarios del producto, entendiendo por usuarios no sólo las personas que interactúan con el producto final, sino aquellas que trabajan en su desarrollo y evolución. Por otro lado, existen características inherentes al producto que sólo son percibidas por profesionales de la informática, como modularidad o legibilidad. Esta diferencia en la detección de la ausencia o presencia de una propiedad en el software nos permite distinguir entre factores de calidad externos e internos. En el primer grupo podríamos incluir básicamente factores como corrección, robustez, extensibilidad, reutilización, compatibilidad, transportabilidad, eficiencia, facilidad de verificación, integridad y facilidad de uso.

1.1.1 Aspectos de calidad del software.

Los factores que determinan la calidad del software pueden ser, pues:

- **Externos:** Los detectados por el usuario del producto: Corrección, robustez, extensibilidad, compatibilidad, eficiencia, portabilidad, facilidad de uso y verificación, integridad, etc..
- **Internos:** Los percibidos sólo por informáticos: modularidad, legibilidad, mantenibilidad, reutilidad, facilidad de verificación formal, etc..

Resulta claro que un mismo factor de calidad puede ser percibido por usuarios y profesionales desde puntos de vista diferentes, y por tanto no pueden enmarcarse únicamente en una clase u otra.

También hay factores contradictorios, especialmente la eficiencia. Mientras mayor eficiencia se desee en el espacio ocupado y en el tiempo de ejecución, menor será la legibilidad, y por tanto menor la facilidad de mantenimiento, extensibilidad, etc.. No obstante, a pesar del caso especial de la eficiencia, suele existir una relación directa entre ambos tipos de factores, de manera que para conseguir los factores externos, los internos deben estar presentes.

Veámoslos someramente uno por uno:

CORRECCIÓN. (Externo)

Es la capacidad del producto software para realizar justamente aquello para lo que fue creado, tal como quedó definido en los documentos de especificación y requerimientos; en ningún caso menos.

ROBUSTEZ. (Externo)

Es la capacidad del producto software para poder funcionar incluso en condiciones fuera de lo normal. Un caso típico son los Sistemas en tiempo real, y los Sistemas tolerantes a fallos. P. ej., cuando en Windows aparece el *Error de protección general* se trata de una falta de robustez, puesto que, aunque el programa no se rompe, sí impide proseguir la tarea, sin proponer ninguna solución ni intentar auto-solucionar el problema por sí mismo. Esto es inadmisible p.ej. en un sistema de piloto automático en una aeronave.

Supóngase el estupor que le daría a un piloto de combate que en mitad de una lucha sin cuartel, el sistema de control de misiles se parase por un error interno con el mensaje: **Error 23**.

FIABILIDAD. (Externo)

Es el grado en que un programa lleva a cabo sus funciones esperadas con la precisión requerida.

Depende del uso que se le vaya a dar. No debe ser igual la fiabilidad de un sistema de lanzamiento de misiles militares, o de seguimiento de una órbita para un satélite, que el cálculo del espacio útil en un edificio según un determinado plano. Por tanto, la fiabilidad depende principalmente de las especificaciones, y principalmente de la corrección. También de la robustez, con objeto de que al usuario no se le rompa el programa ni le aparezcan mensajes raros.

EXTENSIBILIDAD. (Externo)

Es la posibilidad de cambios y adiciones respecto de las especificaciones iniciales.

REUTILIDAD. (Interno)

Es la capacidad de los productos software para ser reutilizados, en su totalidad o en parte, en otras aplicaciones, con la idea de evitar soluciones redundantes a problemas que ya se han solucionado con anterioridad.

Así, un programa debe agrupar en una serie de módulos aislados los aspectos dependientes de la aplicación particular, mientras que las utilidades de que hacen uso deben ser lo suficientemente generales como para ser utilizadas tal cual por otros productos. P.ej., el módulo de apertura de un fichero en WordPerfect, se puede reutilizar en un programa de dibujo como pueda ser el WPDraw.

De hecho, los programas escritos en lenguajes visuales (Visual C++, Visual Basic, Delphi, OSF Motif), intentan explotar esto al máximo utilizando grandes librerías comunes: de cajas de diálogo, de botones, barras de desplazamiento, y un sinfín más de características que hacen que la mayoría de los programas bajo Windows tengan un interfaz común. Para ello, hacen uso de eventos y objetos, que constituyen un pasomás avanzado en la programación, pero que carecen de los formalismos matemáticos de los tad'es que permiten asegurarse completamente de la corrección.

COMPATIBILIDAD. (Externo)

Es la facilidad de los programas para combinarse entre sí. También se puede ver como la posibilidad de usar los resultados de un programa como entrada a otro.

La clave de la compatibilidad es la homogeneidad en el diseño y un consenso en las convenciones sobre estandarización para las comunicaciones entre programas; en definitiva, la reutilización de los formatos de bloques de información. Esta es una de las principales ventajas

del entorno Windows, que inicialmente permitía el transporte de información de una aplicación a otra a través del portapapeles; esto se vio potenciado enormemente con la incorporación de la tecnología OLE.

EFICIENCIA. (Externo e Interno)

Consiste en hacer el mejor uso posible del espacio de memoria disponible, a la vez que se consiguen las ejecuciones más rápidas. La eficiencia también es un aspecto interno, ya que hay algunos aspectos, (como algunos casos de ocupación temporal de memoria) que no pueden ser detectados por el usuario.

PORTABILIDAD. (Externo)

Es la posibilidad de pasar de un programa de un ordenador a otro diferente. También se puede entender como la posibilidad de pasar un programa diseñado bajo un S.O., a otro, lo cual es mucho más difícil cuanto más pronunciadas sean las diferencias entre ambos S.O. (piénsese p.ej. en el traspaso de un programa bajo DOS, al entorno gráfico de Windows). Para ello, es conveniente que lols programas no hagan uso directo de las características propias de cada máquina o S.O..

VERIFICABILIDAD. (Externo e interno)

Desde el punto de vista externo, es la facilidad para preparar procedimiento de aceptación de datos, y en general procedimientos para detectar fallos durante las fases de comprobación y verificación. Depende principalmente del interfaz definido en la especificación formal del componente software, que comporta la vertiente interna de la verificación

SEGURIDAD. (Externo)

Es la capacidad del software para proteger sus diferentes componentes (programas, datos y documentos) contra accesos no autorizados.

INTEGRIDAD. (Externo)

Está muy relacionado con la robustez. El programa no debe corromperse por entradas de datos excesivamente largas, o excesivamente complejas.

FACILIDAD DE USO. (Externo)

Es la facilidad para aprender a usar el programa: facilidad a la hora de introducir datos, a la hora de interpretar resultados, recuperar errores en el uso, etc.. Para ello son muy útiles los GUI (Interfaz Gráfica de Usuario).

-----·o·-----

Entre todos estos factores debemos destacar los cinco primeros (corrección, robustez, extensibilidad, reutilidad y compatibilidad) como claves, ya que reflejan las dificultades más serias en el desarrollo del software. No obstante, la eficiencia debe ser lo suficientemente satisfactoria como para no hacer prohibitivo el uso de un programa.

Si nos centramos en los factores de Extensibilidad y Reutilización, vemos que el primer paso a seguir para alcanzarlos es la *modularización* del software (factor de calidad interno).

La modularización se puede entender como la construcción de grandes programas por medio de la unión de pequeñas piezas de código, llamadas **subrutinas**, que suponía unprimer paso de la programación estructurada avanzada. No obstante, el principal problema de las subrutinas es que carecen de una interfaz de E/S claramente definido, y, en cualquier caso, no poseen los controles de validación y coherencia de tipos entre parámetros formales y reales en las llamadas. Por tanto, esta técnica no implica beneficios grandes a menos que los módulos sean autónomos, coherentes y estén organizados en arquitecturas robustas. Deben tenerse en cuenta criterios como la capacidad de composición y descomposición que poseen los módulos, la comprensión y legibilidad que ofrece cada módulo por separado, la protección, y la estabilidad (pequeños cambios en los requerimientos no deben involucrar un gran cambio en el diseño), etc..

1.2 TIPOS ABSTRACTOS DE DATOS.

TIPO DE DATOS: Es un conjunto de estados o valores permitidos, junto con las operaciones que trabajan sobre tales estados, cuando pertenecen a ese tipo concreto.

Esto es así porque un mismo estado puede pertenecer a tipos diferentes; p.ej., el nº 4, puede ser un entero, un real, un complejo, un racional o un natural.

El usuario puede construir sus propios tipos de datos en un lenguaje determinado, mediante las construcciones sintácticas que se le den para ello.

Así, las estructuras de datos se crean usando constructores predefinidos, (de especial utilidad son los registros y matrices), junto con los tipos que suministra el lenguaje: REAL, INTEGER, CARDINAL, BOOLEAN, etc.. Los tipos de datos definidos por el usuario también se pueden usar para crear otras estructuras de datos más complejas. Ej.:

```
TYPE
        Persona =       RECORD
                                Nombre : ARRAY [0..23] OF CHAR;
                                Apellidos : ARRAY [0..29] OF CHAR;
                                DNI : ARRAY [0..9] OF CHAR;
                        END;
VAR
        Censo : ARRAY [1..1000] OF Persona;
```

La variable Censo es una estructura más compleja formada sobre otra (Persona), definida por el usuario.

Un ***TIPO ABSTRACTO DE DATOS*** permite la construcción de casi cualquier tipo de datos, atendiendo sólo a criterios de comportamiento, y sin relacionarlo para nada con la representación subyacente que nos permita su puesta en ejecución según las funcionalidades particulares que un lenguaje de programación dado nos pueda suministrar. *Su objetivo es hacer referencia a una clase de objetos (conjunto de estados) definidos por una **especificación** independiente de la representación. La implementación del tipo se oculta al usuario del programa.*

La diferencia entre Tipo de Datos y T.A.D. es que el TAD se centra sólo en el comportamiento, no en la implementación. Es algo así como una caja negra. Dice el **qué** y no el **cómo**, aunque por supuesto, el qué tiene que ser viable. Así, se pueden utilizar diferentes cómos,

dando al programador la potestad para seleccionar el que más le convenga, pero sin alterar en ningún momento el comportamiento del TAD, ni, por supuesto, su interfaz con el exterior, que será siempre el mismo, (ya que formalmente se trata del mismo tipo de datos). Así pues, la separación de la especificación de la implementación de un TAD permitiría la existencia de varias implementaciones equivalentes, dándole al programador la posibilidad de cambiar la implementación sin que esto tenga ningún efecto en el resto del sistema.

Juegan un papel muy importante en el desarrollo de software seguro, eficiente y flexible, ya que cuando usamos un TAD definido por otro programador, podemos concentrarnos en las propiedades deseadas de los datos, y en la funcionalidades que nos suministran. Por cuestiones de Correctitud y fiabilidad, es muy importante la correción formal de un TAD; también hay que prestar especial atención a la definición formal en el caso de precondiciones anómalas por cuestiones de Robustez.

Nosotros vamos a utilizar una notación formal para especificar el comportamiento de los TADes.

La primera ventaja que se consigue usando TADes en el diseño de programas es que conllevan el conseguir programas estructurados y modulares. Este tipo de modularidad asegura que las tareas lógicas y los datos se agrupan para formar un módulo independiente cuya especificación funcional es visible a otros programas que lo usan. Estas aplicaciones no necesitan información detallada de cómo están implementadas las tareas y los datos. Esto hace que podamos implementar estas operaciones y tipos de datos de diferentes formas, siempre que se ajusten a las especificaciones. Esta separación de especificaciones e implementación asegura que muchas decisiones de diseño, como eficiencia, consenso entre velocidad y ocupación de memoria, etc., puedan tomarse en una etapa posterior, cuando se ha probado que las propiedades lógicas de los tipos de datos y sus operaciones satisfacen la aplicación. Para todo ello es muy útil la **compilación separada**, que nos permite verificar la corrección de los interfaces definidos, antes de proceder a la implementación propiamente dicha de los TADes.

1.3 FORMALISMOS PARA LA ESPECIFICACION DE T.A.D.ES.

Las propiedades de los tipos INTEGER o BOOLEAN son bien conocidas. Sin embargo, en el momento en que intervienen otros tipos de datos, como pilas, listas, árboles o grafos, cuyo comportamiento y propiedades no son tan bien conocidos, es necesario disponer de definiciones adecuadas para las operaciones sobre dichos tipos, y, en general, sobre cualquier otro que se vaya a utilizar en un programa, de manera que se permita:

- Usar dichas operaciones en los diseños, o sea, poder usarlos aunque se carezca de la implementación.
- Establecer precondiciones y postcondiciones sobre las estructuras de ese tipo, con objeto de fijar el comportamiento de los programas que hacen uso de ellos, y verificar los diseños que se deben ajustar a dichos comportamientos.

Para ello, es necesario emplear una herramienta matemática cuya exactitud no dé lugar a ambigüedades a la hora de entender el funcionamiento de una determinada operación.

La especificación formal más ampliamente extendida es la ***especificación algebraica***, que constituye un caso particular de las especificaciones axiomáticas, y que usaremos de forma intensiva a lo largo del curso.

El concepto básico es la **presentación**, que no es más que un compendio de símbolos (al que se llama **signatura**; estos símbolos representan operaciones y combinaciones entre ellas), que se pueden combinar de una forma determinada , y un conjunto de ecuaciones que permiten especificar la relación de equivalencia entre los símbolos obtenidos en base a la signatura, lo que da lugar a un conjunto cociente formado por una serie de clases de equivalencia, que agrupan a los términos lexicograficamente distintos, pero semánticamente iguales.

Así, para definir un tipo T se tendrán una serie de operaciones, de la forma:

$$f: T_{e1} \times ... \times T_{en} \longrightarrow T_{s1} \times ... \times T_{sm}, \text{ con } n,m \geq 0,$$

que representa la signatura (convención de signos a emplear); ej.:

Suma: $N \times N \longrightarrow N$
OR : $B \times B \longrightarrow B$

donde **N** y **B** son los Dominios, que pueden ser diferentes del TAD que estemos definiendo; ej.:

Re, Im: $C \longrightarrow R$

C representaría a los números Complejos, y **R** a los reales.

Las ecuaciones que definen el comportamiento de las operaciones, en base a variables pertenecientes a un cierto dominio. Ej.:

OR(a, b) == NO(AND(NO(a), NO(b)))

El papel de las ecuaciones es establecer identificaciones entre términos lexicográficamente distintos, pero que denotan el mismo valor semántico, de manera que los términos equivalentes se puedan *reducir* a una única **forma canónica**, representante de una clase de equivalencia.

Por tanto, más que ecuaciones, podríamos considerarlas reducciones, pues su utilidad principal es llegar a formas canónicas de representación de los diferentes estados del TAD; tales reducciones se efectúan siempre de izquierda a derecha. No obstante, estas ecuaciones también nos servirán para hacer demostraciones por inducción; en tales casos no distinguiremos entre parte izquierda y parte derecha de la ecuación.

Veamos un ej. para entender esto. Sea la siguiente especificación formal para describir el comportamiento de los números Naturales:

tipo N (*Naturales*)
Operaciones
0: $\longrightarrow$ N
suc: N $\longrightarrow$ N
suma: N × N $\longrightarrow$ N
Ecuaciones a,b:N
suma(suc(a), b) == suc(suma(a, b)) <<1>>
suma(0,b) == b

Esta especificación por sí misma no es nada. Ahora bien, nos damos cuenta de que podemos establecer una biyección entre los términos canónicos de la misma, y los números

naturales.

Así pues, las formas canónicas son: **0, suc(0), suc(suc(0)), suc(suc(suc(0))),** etc., que podemos asociar a los números naturales 0, 1, 2, 3, etc.. Nótese que la especificación de los números naturales pares sería exactamente igual: 0-0, suc(0)-2, suc(suc(0))-4, etc.. O sea, por un lado tenemos que distinguir entre los garabatos que escribimos, y los conceptos que queremos representar. La potencia aparece cuando podemos establecer una relación entre ambos elementos, ya que los garabatos y las reglas formales por las que se rigen, nos permiten descubrir y demostrar propiedades de los conceptos.

Las álgebras que usaremos tienen las siguientes características:

- Son álgebras generadas por términos (los estados se representan por aplicaciones sucesivas de diferentes funciones u operaciones). Las operaciones que conforman las formas canónicas se llaman **operaciones generadoras**. En el ej. anterior, las funciones generadoras serían la *función constante 0*, y la *función suc()*.

- Son álgebras con el mayor número posible de valores (términos distintos), mientras no se diga lo contrario a través de las ecuaciones.

Si en la especificación anterior no hubiésemos indicado <<1>>, se tendría que:

suma(0, suc(0)) ≠ suc(0)

lo cual es incorrecto con respecto a la semántica que queremos dar a esta presentación.

Las especificaciones que se obtienen de este modo se dice que definen **tipos abstractos de datos** (TADes); las implementaciones que de ellos se hagan serán **tipos de datos** concretos.

1.3.1 Estilo constructivo.

Para la creación de los estados o valores que puede tener un TAD, vamos a considerar el estilo constructivo, y *tales valores los vamos a generar* a partir de la presentación, *como la aplicación sucesiva de un subconjunto de las funciones definidas, a las que vamos a llamar funciones generadoras del tipo en cuestión*. En el caso de los números naturales, las funciones generadoras son la función constante 0, y la primitiva suc().

Entre las operaciones que soporta un TAD, hay que diferenciar entre las **primitivas** (o de enriquecimiento), que son inherentes al propio tipo, y cuya especificación es imposible si no se conoce la construcción de los términos finales o formas canónicas, y el resto de operaciones (llamadas **derivadas**), que puede construirse sólo a través de primitivas. Esta distinción es muy importante a la hora de implementar el tipo. Una operación primitiva debe formar parte obligatoriamente del módulo de programa que conoce la estructura de datos interna que representa al tipo; sin embargo una operación derivada puede formar parte de una biblioteca de nivel superior, que desconozca dicha estructura, y que opere exclusivamente a través de las operaciones que suministra el módulo del tipo. Algunas veces, por cuestiones de eficiencia, las operaciones derivadas también se incluyen en el módulo principal (en lugar de en una biblioteca separada), para que hagan un uso más efectivo de las estructuras que componen al tipo.

A veces, las operaciones derivadas también se incluyen como parte del TAD, con objeto de aumentar la potencia del juego de operaciones suministrado por el TAD.

Así, el método constructivo de un tipo T consiste formalmente en:

- Dividir las operaciones en:

* Constructores o Constructivas: Aquellas que como resultado dan un valor de tipo T (denotaremos por T el Dominio del TAD que estamos definiendo), y, posiblemente información de tipo adicional.

Caso especial son las constantes, en las que no hay dominio de partida.

Las generadoras son un caso particular de los constructores.

* Selectores: Partiendo de un valor de tipo T, y posiblemente información adicional, producen información relativa a la entrada, pero en ningún caso esta salida deberá ser del tipo T.

* Auxiliares: Tienen la misma forma que el resto de operaciones de la signatura. Sin embargo,las pondremos agrupadas aparte para indicar que sólo sirven de apoyo a la especificación, no tienen porqué implementarse, y, sobre todo, el usuario no tendrá conocimiento de ellas en ningún caso, ya que le dan información innecesaria que puede poner en peligro la integridad del tipo.

- Buscar G_T (conjunto de operaciones generadoras), de manera que cualquier valor del TAD tenga una representación en base a f perteneciente a G_T, muy probablemente aplicadas reiteradamente un número finito de veces. (Los naturales se obtienen como aplicación reiterada de suc() sobre la constante 0.

Así, el resultado de cualquier otra operación, que tenga como recorrido a T, se tiene que poder representar sólo en base a funciones de G_T. Los términos generados de esta forma pueden denotar siempre valores distintos, con lo que G_T, sería una *familia libre de generadores*; o puede ocurrir que varios términos denoten un mismo valor del tipo T. O sea, cuando aplicaciones sucesivas de funciones de G_T dan lugar a valores siempre diferentes (irreducibles, esto es, no es posible efectuar reducciones de ningún tipo), se dice que G_T es una **familia libre de Generadores**. Véanse p.ej. los números enteros.

Para cada tipo es muy importante determinar un G_T lo más reducido posible (y libre, lo cual en algunos casos es imposible), pues de esta forma se simplifica el número y complejidad de las ecuaciones que definen las operaciones primitivas.

A los generadores no se les asocia ninguna información semántica (no aparecen a la izquierda de ninguna ecuación), a no ser que la familia de que forman parte no sea libre, como en el caso de los número enteros, donde los generadores son 0, suc() y pred(), y a pesar de ello se tienen las ecuaciones:

pred(suc(n)) == n
suc(pred(n)) == n

El resto de las operaciones se deberán especificar por completo de forma constructiva, en base a los generadores (deben dar lugar siempre a una forma canónica, constituida sólo por funciones generadoras).

Así, cada valor del TAD debe tener una única representación en base a funciones de G_T.

Por otro lado, las presentaciones que establezcamos deben ser completas y correctas:

- Completas: Cuando se define una función nueva sobre el TAD en cuestión, hay que

considerar todas las formas canónicas que ese TAD pueda tomar. Así, en el caso de los naturales, no podríamos haber desechado <<1>>, porque es donde consideramos el único valor de TAD que no empieza por suc(...).

Esta completitud viene derivada de la semántica asociada al tipo que estamos definiendo. La completitud también implica que tras aplicar una operación a una forma canónica, se pueda reducir siempre a otro término formado exclusivamente por generadores.

- Correctas (Consistente): Una misma operación no puede dar lugar a dos formas canónicas diferentes, y por tanto semánticamente diferentes, ya que según hemos dicho, estamos ante álgebras en las que todos los términos son diferentes mientras no se demuestre lo contrario (mientras no se puedan reducir a una forma común mediante las ecuaciones). O sea, no posee contradicciones.

Ej.:

suma(suc(a), b) == suc(suma(a,b))
suma(a, suc(b)) == suc(suc(suma(a,b)))
suma(0, 0) == 0
suma(a,b) == suma(b,a)

Con las tres primeras ecuaciones, estamos diciendo que el comportamiento de la operación suma es: suma(a, b)=a+2b. Por tanto, especificar la propiedad simétrica (a través de la 4ª ecuación), establece que para cualquier a, b: a+2b=b+2a, lo cual puede ser verdad en algún álgebra que nos inventemos, pero resulta un disparate cuando hablamos del comportamiento de los números naturales.

Nótese la importancia de las constantes, ya que las formas canónicas se forman de manera recursiva, y como en toda recursión, es necesario que se produzca un paso final, o definición directa que dé un valor base sobre el que construir ff.cc. más complejas mediante vuelta atrás (backtracking).

A poco que el tipo que estemos definiendo se complique, aparecerán no sólo operaciones totales (definidas para cualquier valor de su dominio), sino también **operaciones parciales**, definidas únicamente para ciertos valores de su dominio, valores que en general quedan englobados en un subconjunto del dominio, que cumple ciertas propiedades que quedan identificadas a través de un predicado. Así, en un nuevo apartado de nuestras especificaciones, indicaremos las **precondiciones** o predicados que deben satisfacerse para poder aplicar una de estas operaciones parciales. En la definición de ecuaciones que explique el comportamiento de esa función parcial se omitirán aquellas formas canónicas que se salen fuera del dominio de la operación parcial de que se trate. Ej.:

pre resta resta(m, suc(s)) : not (m=0)
resta(m, 0) == m
resta(suc(m), suc(s)) == resta(m, s)

1.3.2 Especificaciones de algunos tipos simples.

Vamos a estudiar a continuación los siguientes tipos simples:

- Booleano (Lógico).

- Número natural.
- Número entero.
- Número complejo.

1.3.2.1 BOOLEANO.

El tipo correspondiente al álgebra de Boole de los valores lógicos, se puede especificar de manera directa dando las constantes V y F, que constituyen por sí solas un conjunto libre de generadores. el resto de las operaciones se definen en función de estas constantes. El resto de operaciones se define en función de estas constantes.

tipo B (* Booleano *)
generadores
V, F: $\longrightarrow$ B
constructores
not: B $\longrightarrow$ B
and, or: B × B $\longrightarrow$ B
->, <->: B × B $\longrightarrow$ B
ecuaciones, b:B
not(V) == F
not(F) == V
and(V, b) == b
and(F, b) == F

(1) $\left.\begin{array}{l} or(V,b)\equiv V \\ or(F,b)\equiv b \end{array}\right\} or(a,b)\equiv not(and(not(a),not(b)))$ (2)

->(a,b) == or(not(a), b)
<->(a, b) == and(->(a, b), ->(b, a))

NOTA: Si optamos por la opción (2), entonces las operaciones **not** y **and** son operaciones primitivas, o sea, que se hallan definidas directamente en función de los términos finales o formas canónicas, que en este caso son solamente dos constantes. La opción (2) no es más que la equivalencia: $a\cup b\equiv\overline{\overline{a}\cap\overline{b}}$.

Así, **or**, **->** y **<->** son operaciones derivadas de las anteriores.

Obsérvese que en este caso, todos los valores del tipo (Booleano en nuestro caso) están definidos mediante operaciones constantes. Hay muchas situaciones en las que los valores se especifican mediante una secuencia enumerada de identificadores u operaciones constantes, como puede ocurrir por ejemplo con los días de la semana. En estos casos, cada valor independiente vendrá dado por una operación constante cuyo dominio es, por tanto, inexistente, y cuyo rango es el del tipo que se define. El caso de los números de la semana quedaría:

tipo Día
dominios B
generadores
L, M, X, J, V, S, D: $\longrightarrow$ Día
selectores
EsFinDeSemana: Día $\longrightarrow$ B

ecuaciones

EsFinDeSemana(L) == F
EsFinDeSemana(M) == F
EsFinDeSemana(X) == F
EsFinDeSemana(J) == F
EsFinDeSemana(V) == F
EsFinDeSemana(S) == V
EsFinDeSemana(D) == V

Es interesante observar como a medida que la cantidad de valores enumerados crece, aumenta el número de ecuaciones necesarias para efectuar la especificación, ya que si los valores no pueden especificarse de forma recursiva, tampoco puede especificarse recursivamente el comportamiento de las operaciones que trabajan con ellos. P.ej. si queremos especificar una operación que nos dice si dos días son iguales o no, se tendría:

selectores

=: Día × Día ⟶ B

ecuaciones, a, b ∈ Día

=(a, b) == =(b, a)

=(L, L) == V	=(M, V) == F	=(J, S) == F
=(L, M) == F	=(M, S) == F	=(J, D) == F
=(L, X) == F	=(M, D) == F	=(V, V) == V
=(L, J) == F	=(X, X) == V	=(V, S) == F
=(L, V) == F	=(X, J) == F	=(V, D) == F
=(L, S) == F	=(X, V) == F	=(S, S) == V
=(L, D) == F	=(X, S) == F	=(S, D) == F
=(M, M) == V	=(X, D) == F	=(D, D) == V
=(M, X) == F	=(J, J) == v	
=(M, J) == F	=(J, V) == F	

Véase como la primera ecuación denotra la propiedad conmutativa de la operación = que estamos definiendo. De no indicar esta ecuación, y si queremos que esta operación sea coherente con nuestro concepto de igualdad entre los días de la semana, sería necesario no sólo indicar, p.ej., que =(L, V) == F, sino también que =(V, L) == F; y así con todas las demás. Vemos pues, que la propiedad conmutativa nos evita la especificación de montones de ecuaciones similares.

1.3.2.2 NUMERO NATURAL.

Especificamos el tipo correspondiente al álgebra de los números naturales por recursión, partiendo de los generadores 0 y suc(), con los que se definen todos los valores distintos del tipo, y enriquecemos con las operaciones aritméticas y las relaciones de orden e igualdad.

tipo N (* Natural *)
dominios N, B
generadores

0: ⟶ ℕ
suc: ℕ ⟶ ℕ

constructores

suma, producto : $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$
resta: $\mathbb{N} \times \mathbb{N} \not\longrightarrow \mathbb{N}$
selectores
<=, = : $\mathbb{N} \times \mathbb{N} \longrightarrow$ B
precondiciones m,s $\in \mathbb{N}$
resta(m, s): <=(s, m)
ecuaciones m,n:$\mathbb{N}$
<=(0, n) == V
<=(suc(n), 0) == F
<=(suc(n), suc(m)) == <=(n, m)
=(n, m) == and(<=(n, m), <=(m, n))
suma(0, m) == m
suma(suc(n), m) == suc(suma(n, m))
producto(0, m) == 0
producto(suc(n), m) == suma(m, producto(n, m))
resta(n, 0) == n
resta(suc(n), suc(m)) == resta(n, m)

NOTA: Aquí se observa la aparición de una función u operación parcial resta, que no está definida para cualquier conjunto de valores de sus argumentos. Para dejar constancia de cuando se puede proceder a su cálculo, se incluye un apartado de precondiciones, en el que se indica el aserto (operación que se debe verificar siempre), antes de la ejecución de esa operación. A tal aserto se le llama **precondición**.

Aunque no lo usaremos por ahora, también es muy útil el concepto de **postcondición**, que no es más que un aserto para después de la ejecución de una determinada operación.

1.3.2.3 NUMERO ENTERO.

Especificamos el tipo correspondiente al álgebra de los números enteros por recursión, partiendo de la constante 0 y de las operaciones suc y pred, que definen todos los valores distintos del tipo y cuya combinación no genera valores nuevos, por lo que, a diferencia de los que ocurre con los números naturales, estos generadores no son libres, sino que existe una dependencia entre ellos que figurará entre las ecuaciones de la especificación.

tipo Z
dominios Z, B, $\mathbb{N}$
generadores
0: $\longrightarrow$ Z
suc, pred: Z $\longrightarrow$ Z
constructores
suma: Z × Z $\longrightarrow$ Z
resta: Z × Z $\longrightarrow$ Z
producto: Z × Z $\longrightarrow$ Z
auxiliares
nro_pred_y_suc: Z $\longrightarrow \mathbb{N} \times \mathbb{N}$
suc1, suc2: $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N} \times \mathbb{N}$
selectores
<=, =: Z × Z $\longrightarrow$ B

ecuaciones n,m ∈ Z
(1) pred(suc(n)) == n
(2) suc(pred(n)) == n
resta(n, 0) == n
resta(n, suc(m)) == pred(resta(n, m))
resta(n, pred(m)) == suc(resta(n, m))
suma(n, m) == resta(n, resta(0, m))
producto(n, 0) == 0
producto(n, suc(m)) == suma(n, producto(n, m))
producto(n, pred(m)) == resta(producto(n, m), n)
nro_pred_y_suc(0) == (0, 0)
nro_pred_y_suc(suc(n)) == suc2(nro_pred_y_suc(n))
nro_pred_y_suc(pred(n)) == suc1(nro_pred_y_suc(n))
suc1(n, 0) == (suc(n), 0)
suc1(n, suc(m)) == (n, m)
suc2(0, m) == (0, suc(m))
suc2(suc(n), m) == (n, m)
(3) <=(n, m) == <u><=</u>(nro_pred_y_suc(resta(m,n)))
=(n, m) == and(<=(n, m), <=(m, n))

NOTA: En este ejemplo se introduce el concepto de operación auxiliar. Una <u>operación auxiliar</u> es una funcionalidad (posibilidad de hacer algo) que permite especificar con mayor facilidad una determinada operación.

En este caso surge el problema de que los parámetros de entrada a la función <=:Z×Z ⟶ Z, puede ser cualquier sucesión de funciones suc() y pred(); sin embargo, este par de generadores no son libres, como se desprende de las ecuaciones (1) y (2), o sea, hay ciertas combinaciones entre ellos, que denotan al mismo término o forma canónica.

La operación nro_pred_y_suc():Z ⟶ NxN, devuelve un par de números *Naturales*, que no es más que el número normalizado de operaciones pred() y suc() que constituyen a un Entero (uno de los dos valores será 0). Así pues, la operación <=() de la parte derecha de (3), no es la que estamos definiendo en esta especificación algebraica, sino la que definimos en el apartado anterior para los números naturales. La confusión radica en que poseen el mismo nombre.

1.3.2.4 NUMERO COMPLEJO.

En este apartado vamos a estudiar los números complejos, cuyas características merecen ser comentadas independientemente.

tipo ℂ (* Complejo *)
dominios Z, ℂ
generadores
complejo: Z × Z ⟶ ℂ
constructores
suma, resta, producto: ℂ × ℂ ⟶ ℂ
selectores
real, imaginaria: ℂ ⟶ Z
ecuaciones $r_1, i_1, r_2, i_2 \in Z$
real(complejo(r_1, i_1)) == r_1
imaginaria(complejo(r_1, i_1)) == i_1

suma(complejo(r_1, i_1), complejo(r_2, i_2)) ==
    complejo(suma(r_1, r_2), suma(i_1, i_2))
resta(complejo(r_1, i_1), complejo(r_2, i_2)) ==
    complejo(resta(r_1, r_2), resta(i_1, i_2))
producto(complejo(r_1, i_1), complejo(r_2, i_2)) ==
    complejo (
        resta(producto(r_1, r_2), producto(i_1, i_2)),
        suma(producto(r_1, i_2), producto(i_1, r_2))
)

Como puede observarse, en este caso, existe un único generador (llamado complejo), para obtener cualquier término canónico del tipo $\mathbb{C}$, a diferencia de los casos anteriores, en los que existía un generador constante, el 0, a partir del cual se iban construyendo el resto de los valores mediante la adición de una nueva operación.

Este hecho nos hace poder considerar a cualquier valor de tipo $\mathbb{C}$ como un par de valores de tipo Z, lo cual coincide hasta cierto punto con nuestro concepto de **registro** propio de cualquier lenguaje de programación. Por otro lado, todo tipo registro al que estamos acostumbrados, posee operaciones que permiten:
1.- Construir un registro a partir de valores de cada uno de sus componentes.
2.- Extraer el valor de cada uno de sus componentes.

De esta manera, si consideramos el tipo $\mathbb{C}$ como un registro, la operación **complejo** se encarga del primer punto, mientras que las operaciones **real** e **imaginaria** se encargan de extraer los valores de los componentes, tal y como exige el segundo punto.

Siguiendo este mismo esquema podemos construir cualquier tipo registro que se nos antoje. P.ej., supongamos que deseamos un registro llamado MiRegistro que posee dos campos de tipo boolean, otro campo de tipo natural, y un último campo de tipo complejo. Supongamos que queremos llamar a dichos campos U, V, X e Y respectivamente. La especificación sería:

tipo MiRegistro
dominios B, $\mathbb{C}$, $\mathbb{N}$
generadores
    miRegistro: B × B × $\mathbb{N}$ × $\mathbb{C}$ ⟶ MiRegistro
selectores
    U, V: MiRegistro ⟶ B
    X: MiRegistro ⟶ $\mathbb{N}$
    Y: MiRegistro ⟶ $\mathbb{C}$
ecuaciones $b_1, b_2 \in B$, $n \in \mathbb{N}$, $c \in \mathbb{C}$
    U(miRegistro(b_1, b_2, n, c)) == b_1
    V(miRegistro(b_1, b_2, n, c)) == b_2
    X(miRegistro(b_1, b_2, n, c)) == n
    Y(miRegistro(b_1, b_2, n, c)) == c

Así, la construcción de un registro pasa por crear un generador que admite como parámetros los valores de cada uno de los campos, y por crear tantos selectores como campos haya; cada selector extraerá uno de los valores del registro.

1.4 REQUISITOS DE LOS LENGUAJES PARA LA ESPECIFICACIÓN DE TAD.

La programación estructurada (eliminación de la sentencia go to), la encapsulación de datos (el uso de módulos para agrupar procedimientos relacionados), y la abstracción de datos (el uso de tipos de datos junto con los procedimientos de manipulación correspondientes bien definidos), juegan un papel de fundamental importancia en la producción de software correcto.

1.4.1 Pascal.

Pascal es un lenguaje estructurado que posee algunas facilidades para la abstracción:
- Nos da la posibilidad de ocultar una implementación dentro de procedimientos, así como efectuar desarrollo descendente por refinamientos sucesivos.
- Permite crear nuestros propios tipos de datos:
  * Posibilidad de definir y manipular registros.
  * Posibilidad de definir tipos de datos nuevos a partir de los ya existentes: enumerados, conjuntos, etc..

1.4.2 Modula-2 y Ada.

Añaden a las características del Pascal, las siguientes posibilidades:
- Encapsulamiento y ocultación de la información. Permiten separar los módulos de definición de los de implementación, separando por tanto el **qué** del **cómo**. Así pues, es posible ocultar la implementación de un T.A.D.. Los procedimiento de acceso a un TAD aparecen en el módulo de especificación como cabeceras de procedimientos, sirviendo de **interfaz** con el módulo de implementación.

- Compilación separada. Permite la comprobación (especialmente de tipos), de los interfaces definidos sobre un TAD determinado, antes de proceder a su implementación, evitando rectificaciones innecesarias en los módulos de implementación.

- Incorporan el tipo *procedimiento* que es muy útil en el concepto de **iteradores**. Un iterador es una operación que permite procesar cada uno de los elementos que integran una estructura más compleja. Este procesamiento viene dado precisamente por un parámetro de tipo *procedimiento* que es el encargado de alterar el estado o valor de cada uno de los componentes.

Es necesario indicar en este punto, que los TADes que veremos en capítulos posteriores, consisten únicamente en establecer relaciones más o menos complejas entre elementos de otro tipo llamado **tipo base**. El iterador actúa sobre cada uno de los elementos del tipo base que integran la estructura compleja.

- El tipo opaco, que permite dar al programador la posibilidad de hacer uso de un tipo sin conocer para nada su estructura. En Ada, los tipos opaco pueden ser de varios tipos según se puedan efectuar sobre ellos operaciones como la igualdad, etc..

- Ada también permite el uso de genéricos, que no son más que TADes incompletos en los que falta el tipo base. Para poderlos utilizar, deben ser **instanciados** usando como parámetro un tipo base.

1.4.3 Lenguajes orientados a objetos.

Un paso más en la abstracción de datos, nos lleva a los lenguajes de programación orientados a objetos, tales como Smalltalk, C++, Eiffel, Beta, Java, etc..

En la metodología orientada a objetos, un objeto no es más que un conjunto de datos privados (cuyos valores dan lugar al estado del objeto), y unas operaciones para acceder y procesar los datos.

Algunos conceptos interesantes son:

- Mensaje: Se llama así a la llamada a la operación que permite procesar un objeto.
- Método: Es la implementación oculta de cada operación.
- Herencia. Una clase de objetos puede heredar características y operaciones de otra clase más general.
- Polimorfismo. Un objeto definido de tipo general puede, en realidad, estar referenciando a cualquiera de los que de él heredan.
- Vinculación dinámica. Cuando se envía un mensaje a un objeto polimórfico, se elige en tiempo de ejecución qué versión del método se va a ejecutar.

1.5 COMPLEJIDAD DE LOS ALGORITMOS.

Normalmente hay muchos programas o algoritmos que pueden resolver una misma tarea o problema. Es necesario, por lo tanto, considerar los criterios que podemos usar para determinar cuál es la mejor de las posibilidades en cada situación. Estos criterios pueden incluir características de los programas tales como documentación, portabilidad, etc. Alguno de estos criterios puede ser analizado cuantitativamente, pero en general, la mayoría no pueden ser evaluados con precisión. Puede que el criterio más importante (después de la corrección, por supuesto) es el que es normalmente conocido como eficiencia. Informalmente, la eficiencia de un programa es una medida de los recursos de tiempo y espacio (memoria o almacenamiento secundario) que se requieren en la ejecución. Es deseable, por supuesto, minimizar estas cantidades tanto como sea posible. Antes de poder cuantificar esto es preciso desarrollar una notación rigurosa para poder analizar los programas.

El tiempo y el espacio requeridos por un programa medidos en segundos de tiempo de computador y en número de bytes de memoria es muy dependiente del ordenador usado. Por esto, debemos buscar medidas más simples y abstractas, que sean independientes del ordenador real. Este modelo abstracto puede servir como base para comparar diferentes algoritmos. En la práctica nos interesa saber la forma en que se comporta el tiempo y el espacio empleado en función del tamaño de la entrada.

Objetivos a la hora de resolver un problema:

- Algoritmo fácil de entender, codificar y depurar.
- Uso eficiente de los recursos del computador y, en general, que se ejecute con la mayor rapidez posible.

Para programas que se ejecutan una o pocas veces el primer objetivo es el más importante. Si se va a ejecutar muchas veces, el coste de ejecución del programa puede superar con mucho al de escritura. Es más ventajoso, desde el punto de vista económico, realizar un algoritmo complejo siempre que el tiempo de ejecución del programa resultante sea significativamente menor que el de un programa más evidente. Y aún en situaciones como esa, quizás sea conveniente aplicar primero un algoritmo simple, con objeto de determinar el beneficio real que se obtendría escribiendo un programa más complicado. En la construcción de un sistema complejo, a menudo es deseable aplicar un prototipo sencillo en el cual se puedan efectuar simulaciones y mediciones antes de dedicarse al diseño definitivo.

1.5.1 Medición del tiempo de ejecución de un programa.

El tiempo de ejecución de un programa depende de factores como:

- Los datos de entrada del programa,
- La calidad del código generado por el compilador utilizado para crear el programa objeto,
- La naturaleza y rapidez de las instrucciones de máquina empleadas en la ejecución del programa, y
- La complejidad de tiempo del algoritmo base del programa.

El hecho de que el tiempo de ejecución dependa de la entrada indica que el tiempo de ejecución de un programa debe definirse como una función de la entrada. Por ejemplo, en algoritmos de ordenación, cuanto mayor sea el número de elementos a ordenar, mayor será el tiempo necesario.

Se acostumbra a denominar T(n) al tiempo de ejecución de un programa con una entrada de tamaño **n**. Por ejemplo, algunos programas pueden tener un tiempo de ejecución $T(n)=c \cdot n^2$, donde *c* es una constante. Las unidades de T(n) se dejan sin especificar, pero se puede considerar a T(n) como el nœmero de instrucciones ejecutadas en un computador idealizado.

Para muchos programas, el tiempo de ejecución es en realidad una función de la entrada específica, y no sólo del tamaño de ésta. En este caso se define T(n) como el tiempo de ejecución del peor caso, es decir, el máximo valor del tiempo de ejecución para entradas de tamaño *n*.

También suele considerarse $T_m(n)$, el valor medio del tiempo de ejecución de todas las entradas de tamaño *n*. Aunque $T_m(n)$ parece una medida más razonable, a menudo es engañoso suponer que todas las entradas son igualmente probables. Además, en la práctica casi siempre es más difícil determinar el tiempo de ejecución promedio que el del peor caso, pues el análisis se hace intratable matemáticamente hablando, y la noción de entrada promedio puede carecer de un significado claro. Así pues, se utilizará el tiempo de ejecución del peor caso como medida principal de la complejidad de tiempo, aunque se mencionará la complejidad del caso promedio cuando pueda hacerse de forma significativa.

1.5.2 Notación Asintótica (O y Ω)

Para hacer referencia a la velocidad de crecimiento de los valores de una función se usará la notación conocida como **asintótica** (O).

Ejemplo: Decir que el T(n) (complejidad temporal) de un programa es $O(n^2)$ significa que existen constantes enteras positivas c y n_0 tales que para $n \geq n_0$, se tiene que $T(n) \leq cn^2$.

De hecho, O(f(n)) es un conjunto, y viene definido por:

$$O(f(n)) = \{ g: \mathbb{N} \to \mathbb{R}^+ \cup \{0\} \mid \exists\, c \in \mathbb{R}^+, n_0 \in \mathbb{N}.\ \forall\, n \geq n_0.\ g(n) \leq c \cdot f(n) \}$$

Así decir que T(n) es $O(n^2)$ equivale a $T(n) \in O(n^2)$

Ejemplo: Supóngase T(0)=1, T(1)=4, y en general $T(n)=(n+1)^2$. Entonces se observa que T(n) es $O(n^2)$ cuando $n_0=1$ y c=4; es decir, para $n \geq 1$, se tiene que $(n+1)^2 \leq 4n^2$, que es fácil de demostrar.

Ejemplo: La función $T(n)=3n^3+2n^2$ es $O(n^3)$. Para comprobar esto, sean $n_0=0$ y c=5. Para $n \geq 0$, $3n^3+2n^2 \leq 5n^3$. También se podría decir que T(n) es $O(n^4)$ pero sería una aseveración más débil. Así, buscamos la cota superior más baja para T(n).

Cuando se dice que T(n) es O(f(n)), se sabe que f(n) es una cota superior para la velocidad de crecimiento de T(n), en el peor caso general. Para especificar una cota inferior para la velocidad de crecimiento de T(n) en el mejor caso general, se usa la notación T(n) es Ω(g(n)), que

significa que existe una constante c, y otra n_0 tal que $T(n) \geq c \cdot g(n)$. $\forall n \geq n_0$. Formalmente:

$$\Omega(f(n)) = \{ g: \mathbb{N} \to \mathbb{R}^+ \cup \{0\} \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. g(n) \geq c \cdot f(n) \}$$

Cuando en un algoritmo f(n) se tiene que su $T(n) \in O(g(n))$, y $T(n) \in \Omega(g(n))$, se dice que es del orden exacto de g(n), y se denota por $\Theta(g(n))$.

Ejemplo: $T(n)=n^3+2n^2$ es $\Omega(n^3)$. Con c=1 tenemos $T(n) \geq c \cdot n^3$ para c = 0,1,....

Por lo general no consideraremos las constantes de proporcionalidad, un programa con $O(n^2)$ será mejor que uno con $O(n^3)$; sin embargo, tenemos que considerar el tamaño de la entrada. Si el primer programa tiene $T(n)=100n^2$ y el segundo tiene $T(n)=5n^3$, para entradas pequeñas el segundo será más rápido, y sólo en entradas grandes (n>20) será más rápido el primero.

Es posible que $T(n) \in O(g(n))$ y $T(n) \notin \Omega(g(n))$, como p.ej. en el caso:

$$T(n) = \begin{cases} n^3 & si\ n\ es\ par \\ n^2 & si\ n\ es\ impar \end{cases}$$

resulta evidente que cuando n es par, en el peor caso se tiene $T(n) = n^2$, por lo que $T(n) \in \Omega(n^2)$.

1.6 RECURSIVIDAD.

Como se puede deducir de la formulación que vamos a utilizar para expresar el comportamiento de ciertos tipos de datos, las especificaciones algebraicas son puramente funcionales, esto es, permiten expresar el resultado de una operación tan sólo como el resultado de evaluar o reescribir una única expresión en la que sólo aparecen funciones. La principal diferencia entre este método de especificación y su correspondiente de implementación es que, en este último, el resultado de la operación se expresa como un subprograma, con toda la potencia que ello nos suministra, especialmente en lo referente al concepto iterativo de bucle, que permite ejecutar reiteradas veces un trozo de código.

Es por esto que en las especificaciones algebraicas toma especial relevancia el concepto de recursividad, pues pasa a ser el único método disponible para aplicar reiteradas veces una función sobre un valor o sobre una colección de valores.

No profundizaremos de nuevo en este curso sobre las características fundamentales del concepto de recursividad, ya que esto se ha visto en ocasiones anteriores. De entrada podemos decir que un algoritmo recursivo pretende construir la solución a un problema grande P mediante la composición de soluciones a subproblemas más pequeños s(P), dependientes del primero. Para ello un método recursivo f(P) se compone de los siguientes cuatro pasos:

a) Resolución del caso base o trivial. En un algoritmo recursivo, todo problema debe converger a un problema trivial cuya solución viene dada de forma directa.

b) Descomposición del problema P en uno más pequeño s(P) dependiente del primero, y cuya resolución nos permitirá acercarnos a la solución de P.

c) Llamada recursiva con el problema s(P).

d) Una vez obtenida la solución a s(P), llamémosla f(s(P)), operamos sobre ella para obtener la solución al problema general f(P) = c(P, f(s(P))).

Para comprender esto mejor, veamos como ejemplo el cálculo del factorial:

$$f(n) = n! = \begin{cases} 1 & si\ n=0 \\ n \cdot f(n-1) & si\ n>0 \end{cases}$$

a) Como caso base tenemos cuando n=0. Para tal entrada, la salida es directa: 1, sin

necesidad de llamadas a problemas más pequeños.

b) Si la entrada es un valor de n mayor que 0, ¿cómo calcular el valor de n! si no podemos utilizar explícitamente bucles?. El método parte de que n! puede descomponerse en un problema más pequeño s(n) = (n-1), cuya solución f(s(n)) podemos componer con el propio n para obtener la solución a f(n), de la forma f(n) = c(n, f(s(n))) = n*f(s(n)).

c) Hacemos una llamada para solucionar el problema s(n).

d) Por último hacemos la composición c(n, f(n-1)) para calcular f(n)=n!.

1.6.1 Recursividad final y no final.

Desde el punto de vista de la función de composición c(P, f(s(P))), podemos clasificar las funciones recursivas en dos grandes bloques:

1.- **Recursividad final**, cuando la función c() es trivial, o sea:

$$c(P, f(s(P))) = f(s(P))$$

2.- **Recursividad no final**, en cualquier otro caso.

Aunque parezca que la recursividad final no tiene mucho sentido, existen numerosos casos en los que es aplicable, como puede ser el algoritmo para saber si un elemento está o no en una lista; otro ejemplo de recursividad final es el algoritmo de Euclides para el cálculo de máximo común divisor entre dos números:

```
PROCEDURE mcd(a, b : INTEGER) : INTEGER;
VAR
        c : INTEGER;
BEGIN
        IF a=b THEN
                RETURN a;
        ELSIF a>b THEN
                c := mcd(a-b, b);
                RETURN c;
        ELSE
                c := mcd(a, b-a);
                RETURN c;
        END;
END mcd;
```

Como se observa en este ejemplo, el resultado de la función obedece a dos casos bien distintos, el caso trivial cuando a=b, y el caso recursivo cuando a≠b, momento en que aparece la función c() cuyo valor depende únicamente de la solución al problema más pequeño.

Como ejemplo de recursividad no final tenemos el cálculo del factorial, en el que c() no depende tan sólo de (n-1)!, sino también de **n** en sí.

1.6.2 Técnicas de inmersión.

Como dijimos anteriormente, las especificaciones algebraicas de cualquier operación deben indicarse en forma puramente recursiva; sin embargo hay ciertas operaciones cuyo método de resolución es inherentemente iterativo. Al igual que es posible transformar cualquier programa

recursivo en uno iterativo (de hecho el código máquina de un ordenador sólo puede hacer las cosas de forma iterativa, por lo que un algoritmo recursivo expresado en un lenguaje de alto nivel debe ser traducido a su correspondiente iterativo reconocible por el μp), existen métodos para transformar algoritmos iterativos en recursivos, de forma que puedan ser expresados mediante una especificación algebraica. Es lo que se llama **técnica de inmersión**.

Aunque estas transformaciones pueden llevarse al extremo de obtener un resultado recursivo final o no final según se desee, nosotros sólo veremos el caso más directo: la conversión a recursivo no final, o **inmersión no final**.

El método de transformación de un algoritmo iterativo f() consiste, a grandes rasgos, en definir una función más general g(), (con más general, se quiere decir con más parámetros y/o con más resultados), de forma que con ciertos valores de entrada para estos nuevos parámetros, calcula lo mismo que la función original f().

Así, diremos que hemos hecho una inmersión de f() en g(); f() es la **función sumergida**, y g() la **función inmersora**.

Formalmente, se parte de una función $f(\vec{x})$, con unas precondiciones $Q(\vec{x})$, y postcondiciones $R(\vec{x}, \vec{y})$, y llegar a otra $g(\vec{x}, \vec{w})$ de la forma:

$\{Q(\vec{x})\}$	$\{Q'(\vec{x}, \vec{w})\}$
$f: \vec{x} \longrightarrow \vec{y}$	$g: \vec{x}, \vec{w} \longrightarrow \vec{y}, \vec{z}$
$\{R(\vec{x}, \vec{y})\}$	$\{R'(\vec{x}, \vec{w}, \vec{y}, \vec{z})\}$

donde $\vec{y}$ son los parámetros adicionales de la función inmersora, y $\vec{z}$ sus resultados adicionales; Q, Q', R y R' son precondiciones y postcondiciones. Bajo ciertas condiciones $P(\vec{x}, \vec{w})$, el resultado $g(\vec{x}, \vec{w})|_{\vec{y}}$ es igual a $f(\vec{x})$:

$$Q'(\vec{x}, \vec{w}) \cap P(\vec{x}, \vec{w}) \cap [g(\vec{x}, \vec{w}) = (\vec{y}, \vec{z})] \rightarrow R(\vec{x}, \vec{y})$$

Desgraciadamente, por norma general, el encontrar la función inmersora forma parte de la vertiente artística de la programación.

Un método eficaz cuando nuestro *tad* dispone de una operación de acceso directo a cualquiera de sus elementos (p.ej., una lista posicional), consiste en hacer $\vec{z} = []$, y $\vec{w} =$ [contadores de bucle de la versión iterativa, valores finales de los contadores]. De esta forma, la ecuación f() quedará:

$$f(\vec{x}) == g(\vec{x}, \vec{w}_0, \vec{w}_f)$$

donde $\vec{w}_0$ son los valores iniciales de los contadores de bucle, y $\vec{w}_f$ son las constantes que indican el máximo valor que pueden alcanzar los contadores anteriores. Las ecuaciones de g() detectarán cuando el parámetro $\vec{w}$ toma el valor final del bucle (caso trivial), en cuyo caso se devolverá un valor base sobre el que se construirá el resultado. Para ilustrar un caso fácil, veamos cómo podemos construir la lista suma de dos listas de igual longitud; la versión iterativa de informal sería algo así como:

```
ENTRADA: lista1, lista2.
for i := 1 to longitud(l)
        resultado[i] := lista1[i] + lista2[i]
next
SALIDA: resultado.
```

En nuestro caso la conversión quedaría como:

operaciones
Suma : Lista × Lista $\not\longrightarrow$ Lista
auxiliar
Suma2 : Lista × Lista × $\mathbb{N}$ × $\mathbb{N}$ $\longrightarrow$ Lista
precondiciones
pre : Suma(l_1, l_2) : Longitud(l_1) = Longitud(l_2)
ecuaciones
Suma(l_1, l_2) == $Suma_g$(l_1, l_2, 1, Longitud(l_1))
$Suma_g$(l_1, l_2, i, n) == SI i > n ENTONCES
Crear
SI NO
Insertar($Suma_g$(l_1, l_2, suc(i), n), 1,
Elemento(i, l_1) + Elemento(i, l_2))

Vemos que la única ecuación que implica a Suma(), lo que hace es una transformación a $Suma_g$(), incluyendo dos nuevos parámetros, a saber, el valor inicial del bucle, y el valor final. $Suma_g$ se encarga de llamarse a sí misma incrementando cada vez el valor actual del contador del bucle, hasta llegar a un valor más allá del máximo en cuyo caso devuelve el resultado trivial: la lista vacía. En cada paso no trivial, se obtiene la suma de los elementos situados en la posición i de las listas originales. Como la lista actual está en construcción, dicho resultado se inserta siempre en la cabeza de la lista, con lo que una vez construída del todo, habremos obtenido el resultado deseado.

Como ejemplo más completo, veamos como podemos abordar el cálculo del producto entre dos matrices. Partimos de la siguiente especificación:

tipo Matriz
dominios $\mathbb{N}$
sintaxis
generadores
Crear : $\mathbb{N}$ × $\mathbb{N}$ × $\mathbb{N}$ $\longrightarrow$ Matriz
Insertar : $\mathbb{N}$ × $\mathbb{N}$ × $\mathbb{N}$ × Matriz $\longrightarrow$ Matriz
selectores
Dimensiones : Matriz $\longrightarrow$ $\mathbb{N}$ × $\mathbb{N}$
Elemento : $\mathbb{N}$ × $\mathbb{N}$ × Matriz $\longrightarrow$ $\mathbb{N}$
constructores
Mult : Matriz × Matriz $\longrightarrow$ Matriz
auxiliares
primero, segundo : $\mathbb{N}$ × $\mathbb{N}$ $\longrightarrow$ $\mathbb{N}$
precondiciones $n, n_1, n_2 \in \mathbb{N}$
$m, m_1, m_2 \in$ Matriz
pre Crear(n_1, n_2, n) : (n_1 > 0) AND (n_2 > 0)
pre Insertar(n_1, n_2, n, m) : (n_1 > 0) AND
(n_2 > 0) AND
($n_1 \leq$ primero(Dimensiones(m)) AND
($n_2 \leq$ segundo(Dimensiones(m))

pre Elemento(n_1, n_2, m) : ($n_1 > 0$) AND
($n_2 > 0$) AND
($n_1 \leq$ primero(Dimensiones(m)) AND
($n_2 \leq$ segundo(Dimensiones(m))
pre Mult(m_1, m_2) : (primero(Dimensiones(m_1) = segundo(Dimensiones(m_2)) AND
(segundo(Dimensiones(m_1) = primero(Dimensiones(m_2))

ecuaciones

Insertar(p_{11}, p_{12}, n, Insertar(p_{21}, p_{22}, n_2, m)) ==
SI ($p_{11} = p_{21}$) AND ($p_{12} = p_{22}$) ENTONCES
Insertar(p_{11}, p_{12}, n, m)
SINO
Insertar(p_{21}, p_{22}, n_2, Insertar(p_{11}, p_{12}, n_1, m))
Dimensiones(Crear(n_1, n_2, n)) == n_1, n_2
Dimensiones(Insertar(p_{11}, p_{12}, n, m)) == Dimensiones(m)
Elemento(p_{11}, p_{12}, Crear(n_1, n_2, n)) == n
Elemento(p_{11}, p_{12}, Insertar(p_{21}, p_{22}, n_2, m)) ==
SI ($p_{11} = p_{21}$) AND ($p_{12} = p_{22}$) ENTONCES
n
SINO
Elemento(p_{11}, p_{12}, m))
Mult(m_1, m_2) == Mult2(m_1, m_2, 1, 1)
Mult2(m_1, m_2, p_1, p_2) ==
SI p_2 > segundo(Dimensiones(m_1)) ENTONCES
Crear(Dimensiones(m_1), 0)
SI NO SI p_1 > primero(Dimensiones(m_1))
Mult2(m_1, m_2, 1, suc(p_2))
SI NO
Insertar(p_1, p_2, Prod_parcial(p_1, p_2, m_1, m_2)), Mult2(m_1, m_2, suc(p_1), p_2))
Prod_parcial(p_1, p_2, m_1, m_2) == Prod_parcial2(p_1, p_2, m_1, m_2, 1)
Prod_parcial2(p_1, p_2, m_1, m_2, n) ==
SI n > primero(Dimensiones(m_1)) ENTONCES
0
SINO
Elemento(n, p_1) * Elemento(p_2, n) + Prod_parcial2(p_1, p_2, m_1, m_2, suc(n))

En este caso, hacemos uso de dos contadores: el primero para las columnas, y el segundo para las filas. No es necesario aquí, pasar los valores finales de los contadores, ya que éstos pueden obtenerse fácilmente mediante la operación Dimensiones() aplicada a cualquiera de las dos matrices de entrada. El método que se sigue es el mismo, sólo que con dos bucles. Cuando el bucle externo llega a su fin se retorna una matriz base, cuyos elementos se han inicializado a cualquier valor, p.ej. 0. Cuando es el bucle interno el que llega a su final, se genera una nueva llamada en la que se reinicia el bucle interno, y se incrementa el externo, convergiendo así al caso final.

Si ambos contadores están en los límites adecuados, pasamos a calcular el valor de la posición (p_1, p_2) de la matriz resultante, que se Inserta en el resultado parcial subsiguiente. Este cálculo, dado por la operación Prod_parcial es, de nuevo, inherentemente iterativo, por lo que su especificación se hace recursiva mediante una nueva inmersión, gracias a un solo contador de bucle, que también se inicia a 1.

Como vemos, hemos construído la especificación algebraica (recursiva), de una operación cuya descripción algorítmica se comporta de forma intuitivamente iterativa.

1.6.3 Complejidad en programas recursivos.

Aunque este tema se verá en mayor profundidad en asignaturas posteriores, daremos aquí la formulación básica que expresa la complejidad de los algoritmos recursivos en función de la forma de proceder particular de cada uno de ellos. También veremos la demostración de algunos de estos cálculos.

Las funciones recursivas más simples son aquellas que reducen el problema de forma mínima, o sea, pasan de calcular f(n) a calcular f(n-1), y que desarrollan a lo sumo una llamada interna por cada llamada externa. La complejidad de estas funciones se puede expresar como:

$$T(n)=\begin{cases} k_1 & si\ n=0 \\ T(n-1)+k_2 & si\ n>0 \end{cases}$$

No obstante, podemos hacer nuestros cálculos más generales si suponemos que el tamaño del problema no decrece en una sola unidad en cada llamada interna, sino que, en general, decrece en **b** unidades. Asimismo, supondremos que cada ejecución del algoritmo recursivo tiene una complejidad de $\mathbf{n^k}$, y que no estamos ante recursividad lineal, sino que se hacen **p** llamadas recursivas internas por cada externa. En definitiva, consideraremos que nuestro algoritmo tiene la siguiente estructura:

```
ALGORITMO f(n)
        PARA i1 := 1 A n HACER
                PARA i2 := 1 A n HACER
                        :                       :
                                PARA ik := 1 A n HACER
                                FIN
                        :
                FIN
        FIN
        SI caso_base ENTONCES DEVOLVER
        Llamada nº 1 a f(n-b).
        Llamada nº 2 a f(n-b).
                :
        Llamada nº p a f(n-b).
FIN ALGORITMO.
```

Esta estructura se puede denotar de la forma más general:

$$T(n)=\begin{cases} n^k & si\ 0\le n<b \\ pT(n-b)+cn^k & si\ n\ge b \end{cases}$$

Suponiendo n >> b, obtenemos para T(n) el siguiente sumatorio:

T(n) $= pT(n-b) + cn^k =$

$= p(pT(n-2b)+c(n-b)^k)+cn^k = ...$

$$= p^m T(n-mb) + \sum_{i=0}^{m-1} p^i c(n-ib)^k =$$

$$= \sum_{i=0}^{m} p^i c(n-ib)^k$$

suponiendo que $T(n-mb) = n^k$, o sea, que $0 \le n-mb < b$, o lo que es lo mismo $m = \lfloor n/b \rfloor$. Así, podemos hacer la siguiente transformación:

$$\sum_{i=0}^{m} p^i c \frac{(n-ib)^k}{b^k} b^k = \sum_{i=0}^{m} cb^k p^i \left(\frac{n}{b} - i\right)^k$$

Como $m = \lfloor n/b \rfloor$, se tiene $m \le n/b < m+1$, por lo que

$$cb^k \sum_{i=0}^{m} p^i (m-i)^k \le T(n) < cb^k \sum_{i=0}^{m} p^i (m+1-i)^k$$

Así, ahora distinguiremos dos casos. Actuaremos a grosso modo ya que lo que se desea obtener es el O(f(n)).

1) $p = 1$.

$$O\left(cb^k \sum_{i=0}^{m} p^i (m+1-i)^k \right) = O\left(\sum_{i=0}^{m} (m+1-i)^k \right)$$ por ser cb^k constante, y p=1.

Tenemos que $\sum_{i=0}^{m} (m+1-i)^k = (m+1)^k + (m)^k + (m-1)^k + (m-2)^k + ... + 2^k + 1^k \equiv m+1$ términos

cuyo $\lim_{n\to\infty}$ es $(m+1)^k$ repetido m+1 veces, o sea, $(m+1)^{k+1}$. Como m es función líneal de n, tendremos finalmente que:

si p = 1, entonces $T(n) \in O(n^{k+1})$.

2) $p > 1$.

$$O\left(cb^k \sum_{i=0}^{m} p^i (m+1-i)^k \right) = O\left(cb^k p^{m+1} \sum_{i=0}^{m} \frac{p^i (m+1-i)^k}{p^{m+1}} \right) =$$

$$O\left(cb^k p^{m+1} \sum_{i=0}^{m} \frac{(m+1-i)^k}{p^{m+1-i}} \right)$$

sustituyendo m+1-i por j, tendremos:

$O\left(cb^k p^{m+1} \sum_{j=1}^{m+1} \frac{j^k}{p^j} \right)$, donde la suma $\sum_{j=1}^{m+1} \frac{j^k}{p^j}$ converge a una constante en el $\lim_{n\to\infty}$. Así

$O\left(cb^k p^{m+1} \sum_{j=1}^{m+1} \frac{j^k}{p^j} \right) = O\left(cb^k p^{m+1} c_o\right) = O\left(p^{m+1}\right)$, deduciendo así que si p > 1, entonces T(n)

$\in O\left(p^{\left\lceil \frac{n}{b} \right\rceil}\right)$.

p = 1	$O(n^{k+1})$

p > 1	$O\left(p^{\left\lceil \frac{n}{b} \right\rceil}\right)$

Sin embargo, no todos los algoritmos recursivos reducen el problema original en una cantidad constante. Cuando las llamadas recursivas reducen el problema siguiendo el esquema de división (como p.ej. una búsqueda dicotómica), podemos establecer un esquema similar:

$$T(n)=\begin{cases} cn^k & si\ 1\le n<b \\ pT(\frac{n}{b})+cn^k & si\ n\ge b \end{cases}$$

que puede corresponder a un algoritmo como:

```
ALGORITMO f(n)
        PARA i1 := 1 A n HACER
                PARA i2 := 1 A n HACER
                        :                       :
                                PARA ik := 1 A n HACER
                                FIN
                        :
                FIN
        FIN
        SI caso_base ENTONCES DEVOLVER
        Llamada nº 1 a f(n/b).
        Llamada nº 2 a f(n/b).
                :
        Llamada nº p a f(n/b).
FIN ALGORITMO.
```

Operando igual que en el esquema anterior, obtenemos:

$$T(n)=\sum_{i=0}^{m} p^i c(\frac{n}{b^i})^k \text{ , con } m=\left\lfloor \log_b m \right\rfloor .$$

Podemos hacer la siguiente transformación:

$$T(n)=\sum_{i=0}^{m} p^i c(\frac{n}{b^i})^k = n^k \sum_{i=0}^{m} \frac{p^i c \frac{n^k}{b^{ik}}}{n^k} = cn^k \sum_{i=0}^{m} \left(\frac{p}{b^k}\right)^i .$$

Igual que en el epígrafe anterior resultan varios casos que el lector puede deducir varios casos:

$p < b^k$	$O(n^k)$
$p = b^k$	$O(n^k \log_b n)$
$p > b^k$	$O(n^{\log_b p})$

1.7 ERRORES Y CLÁUSULAS DE APOYO.

1.7.1 Cláusula SI...ENTONCES...SI NO...

A partir de este punto, se plantea el problema de indicar una ecuación en función de un predicado que actúa sobre parte de los argumentos de entrada. Veamos p.ej., la especificación de una operación que nos devuelva el MCD de dos naturales. La forma más intuitiva de hacer esto es mediante una cláusula SI..ENTONCES...SI NO..., de la forma:

MCD: $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$

y cuya especificación sería:

```
MCD(n, m) ==        SI =(n, m) ENTONCES
                          n
                    SI NO  SI <=(n, m) ENTONCES
                                 MCD(n, resta(m, n))
                           SI NO
                                 MCD(resta(n, m), m)
```

El problema radica en ver si efectivamente este tipo de construcción se puede poner en forma funcional, que es el único recurso que nos permite el formalismo algebraico.

En efecto, es posible. Crearemos una operación que tomará tres grupos de parámetros; el primero será una valor lógico, y los otros dos será exactamente del mismo tipo o grupo de tipos; si el parámetro lógico es Verdadero, nuestra operación devolverá el primero grupo de parámetros, y en caso contrario devolverá el segundo. Formalmente, por cada operación de la forma Op: $T_{e1} \times ... \times T_{en} \longrightarrow T_{s1} \times ... \times T_{sm}$, que aparezca en una ecuación en cuya parte derecha se necesite una cláusula SI...ENTONCES...SI NO..., supondremos implícita otra operación de la forma Pr_{Op}: $B \times T_{s11} \times ... \times T_{sm1} \times T_{s12} \times ... \times T_{sm2} \longrightarrow T_{s1} \times ... \times T_{sm}$, definida con las siguientes ecuaciones:

$Pr_{Op}(V, t_{11}, ..., t_{m1}, t_{12}, ..., t_{m2}) == t_{11}, ..., t_{m1}$

$Pr_{Op}(F, t_{11}, ..., t_{m1}, t_{12}, ..., t_{m2}) == t_{12}, ..., t_{m2}$

En el caso del MCD, suponemos un operación Pr_{MCD}: $B \times \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$, y definida:

$Pr_{MCD}(V, n_1, n_2) == n_1$

$Pr_{MCD}(F, n_1, n_2) == n_2$

de manera que, gracias a ella, nuestra ecuación que define al MCD, podría quedar como:

$MCD(n, m) == Pr_{MCD}(=(n, m), n, Pr_{MCD}(<=(n, m), MCD(n, resta(m, n)), MCD(resta(n, m), m)))$

Nótese que esta última ecuación formaliza el significado intuitivo de la cláusula SI...ENTONCES...SI NO.... A partir de ahora emplearemos dicha cláusula por ser mucho más legible, y poderse transformar en todo caso a su correspondiente notación funcional.

1.7.2 TRATAMIENTO DE ERRORES.

Existen determinadas operaciones que, por sus características, no pueden ser utilizadas en todas las condiciones posibles, esto es, existen determinados valores de entrada para los cuales no están definidas; p.ej.: la división por 0, o la resta en los números naturales cuando el sustraendo es mayor que el minuendo. Este hecho da lugar al concepto de **operación parcial**.

En una especificación completa, es necesario indicar cómo se comporta una operación, incluyendo también que su comportamiento en caso de situaciones anómalas es la obtención de

un **valor erróneo**. De esta forma, al conjunto de estados posibles de nuestro tipo, se añade uno más, que es el que representa al valor indefinido, al valor sin sentido, al valor cuya utilización debe ser evitada porque no suministra información útil.

De esta manera, igual que se indica el comportamiento del resto de las operaciones con todos los posibles estados de entrada, y al añadir este nuevo estado, es imprescindible (por completitud de la especificación), el indicar como se comportan las operacines cuando uno de sus parámetros es el valor erróneo.

El problema surge en que el número de ecuaciones que definen al tipo, se dispara, haciendo la especificación difícil de leer, y lo que es peor, desviando la atención del lector de los aspectos efectivamente importantes del comportamiento del tipo, o sea, de las operaciones interesantes de verdad. P.ej., la especificación de la resta debería ser desde un punto de vista puro, de la siguiente forma:

tipo N (* Natural *)
dominios N, B
generadores
0: ⟶ ℕ
suc: ℕ ⟶ ℕ
error_N: ⟶ ℕ
constructores
suma, producto : ℕ × ℕ ⟶ ℕ
resta: ℕ × ℕ ⟶ ℕ
selectores
<=, = : ℕ × ℕ ⟶ B
objeto_erróneo: ℕ ⟶ B
ecuaciones m,n:ℕ
objeto_erroneo)0) == F
objeto_erroneo(error_N) == V
objeto_erroneo(suc(n)) == objeto_erroneo(n)
<=(0, n) == SI objeto_erroneo(n) ENTONCES
error_B
SI NO
V
<=(n, 0) == SI objeto_erroneo(n) ENTONCES
error_B
SI NO
F
<=(suc(n), suc(m)) == <=(n, m)
=(n, m) == and(<=(n, m), <=(m, n))
suma(0, m) == m
suma(suc(n), m) == suc(suma(n, m))
suma(error_N, m) == error_N
producto(0, m) == 0
producto(suc(n), m) == suma(m, producto(n, m))
producto(error_N, m) == error_N
resta(n, 0) == n
resta(suc(n), suc(m)) == resta(n, m)
resta(0, suc(n)) == error_N

resta(n, error_N) == error_N
resta(error_N, m) == error_N

Para evitar este problema, tomaremos algunas convenciones con objeto de simplificar la escritura de la especificación, pero sin que el resultado final se vea alterado:

1.- Todo dominio **T** introducido en la especificación ofrece automáticamente dos operaciones visibles: la constante $\mathbf{error_T}$, y el predicado de corrección convenientemente especificado: **objeto_erroneo: T ⟶ B**.

2.- Las ecuaciones que puedan producir errores, se colocarán en una cláusula especial de precondiciones, en la que se indicará el término, y la precondición que debe cumplir para no ser considerado un objeto erróneo.

3.- Durante la evaluación de términos, los errores se propagan automáticamente; así, no es necesario ecuaciones propagación de la forma $\mathbf{op\_x(error_T) == error_T}$.

4.- El resto de ecuaciones que se indiquen en el apartado de semántica ecuacional, supondremos que tanto la parte derecha como la izquierda, están libres de errores. P.ej.: una ecuación como: **op_x(t) == op_y(t)** equivaldrá realmente a:

op_x(t) == SI not [objeto_erróneo(t)] ENTONCES
op_y(t)
SI NO
$\mathbf{error_T}$.

Siguiendo este criterio, nos quedaría la especificación que hemos visto a lo largo de este capítulo.

EJERCICIOS.

1.- Crear la función Es_Par: ℕ ⟶ B, que devuelva **V** si su argumento es un número par, y **F** en caso contrario.

2.- Crear la función Siguiente_Par: ℕ ⟶ ℕ, que devuelva el número par más cercano y mayor que su argumento. Emplear la estructura SI .. ENTONCES .. SINO ...

3.- Crear la función Neg: Z ⟶ Z, que dado un entero, le cambia su signo. Ej.: Neg(-3) = 3; Neg(8) = -8.

4.- Crear la función Conjugado: ℂ ⟶ ℂ, que dado un complejo, devuelve su conjugado.

5.- Si la operación <=: Z × Z ⟶ B, la hubiésemos definido tal como:

<=(n, m) == <=(nro_pred(resta(m, n)), nro_suc(resta(m, n)))

con

nro_suc, nro_pred: Z ⟶ ℕ

y

nro_suc(0) == 0
nro_suc(suc(n)) == suc(nro_suc(n))
nro_suc(pred(n)) == nro_suc(n)
nro_pred(0) == 0
nro_pred(suc(n)) == nro_pred(n)
nro_pred(pred(n)) == suc(nro_pred(n)

¿qué problema nos podríamos encontrar, a efectos de reducciones, si como resultado de resta(m, n) obtuviésemos el resultado pred(pred(suc(suc(suc(0))))?.
Recordar la existencia de las ecuaciones:

pred(suc(n)) == n
suc(pred(n)) == n.

6.- Si la operación <=: Z×Z ⟶ B, la hubiésemos definido tal como:

<=(n, m) == <=(resta(nro_pred_y_suc(resta(m, n))), 0)

¿qué problema puede aparecer?.
Nótese el tipo que retorna cada operación, p.ej., ¿las operaciones resta() que aparecen en la parte derecha, son ambas la misma operación?.

7.- Crear el tipo abstracto de datos Módulo_8 (M_8), que tenga como valores 0, suc(0), ... $suc^6(0)$, $suc^7(0)$. Los generadores serán 0 y suc(). Especificar las funciones Suma, Producto, Resta: M_8×M_8 ⟶ M_8.

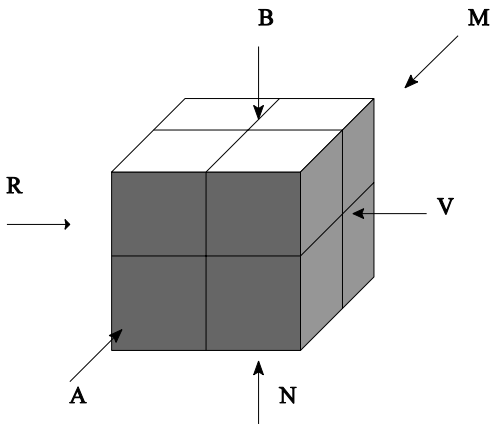
8.- Sea el TAD CCC (Color-Color-Color), que sólo puede tomar uno de los 8 valores siguientes: BVA, BAR, BRM, BMV, NVA, NAR, NRM, NMV. (B-Blanco, N-Negro, V-Verde, R-Rojo, A-Azul, M-Magenta). Crear la función

Todos_Distintos: CCC×CCC×CCC×CCC×CCC×CCC×CCC×CCC ⟶ B

que devuelve **V** si todos sus argumentos son diferentes, y **F** en caso contrario.

Nota: Veamos el significado de esto. Vamos a tener un cubo mágico con los seis colores propuestos. Cada cara estará dividida en 4 trozos (en lugar de 9 como en el caso normal). El TAD CCC representa los colores que tiene cada una de las 8 esquinas que forman el cubo. Vamos a numerar estas esquinas de la forma siguiente:

La función Todos_Distintos nos da las posiciones del cubo que son válidas: en un cubo, no puede haber dos esquinas iguales.



9.- Crear la función

Rotar_Superior: CCC×CCC×CCC×CCC×CCC×CCC×CCC×CCC ⟶
CCC×CCC×CCC×CCC×CCC×CCC×CCC×CCC

de manera que coja sus 4 primeros elementos, y los rote a derecha. Ej.:
Rotar_Superior(BVA, BAR, BRM, BMV, NVA, NAR, NRM, NMV) daría como resultado: BMV, BVA, BAR, BRM, NVA, NAR, NRM, NMV.

Crear análogamente la función Rotar_Inferior, que rotaría los 4 últimos elementos.

Crear la función Hecho: CCC×CCC×CCC×CCC×CCC×CCC×CCC×CCC ⟶ B, que devuelva **V** si sus parámetros son BVA, BAR, BRM, BMV, NVA, NAR, NRM, NMV, en ese orden.

10.- Supongamos la existencia de un tipo **Color**, que posee las constantes: *Blanco, Negro, Azul, Amarillo, Rojo, Verde, Magenta*. En base a este tipo, vamos a implementar unas cuantas operaciones del famoso juego Mastermind, (parecido al Lingo de la TV). En este juego, un jugador propone secretamente una secuencia de cinco colores diferentes, mientras que el jugador contrario debe acertar la combinación correcta. Para ello, lanza a su interlocutor una serie de tentativas. Éste último, ante cada tentativa indica el número de colores que se hallan en la misma posición en la jugada tentativa y en la jugada secreta. Asimismo, indica también qué colores se han acertado, pero cuya posición no coincide plenamente con la de la jugada secreta. P. ej., si la jugada secreta es

Blanco **Negro** **Rojo** **Azul** **Amarillo**

Si el otro jugador hace la tentativa:

Blanco^{+} *Verde* *Rojo*$^{+}$ *Negro** *Magenta*

el resultado será:

2 colores en la misma posición^{+}.

1 color existente no en la misma posición*.

Especificar las operaciones:

Jugar: Color × Color × Color × Color × Color ⟶ Jugada

Es_Válida: Jugada ⟶ Lógico

Resultado: Jugada × Jugada ⟶ $\mathbb{N} \times \mathbb{N}$

Supondremos que el tipo Color dispone de una operación que nos dice si dos colores son iguales o diferentes.

La operación Jugar la supondremos generadora del tipo Jugada; la operación Es_Válida nos dice cuando una jugada posee todos sus colores diferentes o no; y Resultado nos da el número de aciertos completos (mismo color, misma posición), y el número de aciertos aproximados (mismo color, diferente posición).

11.- Crear el tipo **Polinomio**,en el que tanto los coeficientes como los exponentes son números naturales, o sea, de la forma:

$p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + ... + a_n x^n \qquad n, a_i \in \mathbb{N}$

con las operaciones:

cero: ⟶ Polinomio

añadir: Polinomio × $\mathbb{N}$ × $\mathbb{N}$ ⟶ Polinomio

coeficiente : Polinomio × $\mathbb{N}$ ⟶ $\mathbb{N}$

evaluar: Polinomio × $\mathbb{N}$ ⟶ $\mathbb{N}$

sumar: Polinomio × Polinomio ⟶ Polinomio

Crear cuantas operaciones auxiliares se necesiten.

12.- Añadir al tipo anterior, la operación de producto entre polinomios:

producto: Polinomio × Polinomio ⟶ Polinomio

creando, análogamente, cuantas operaciones auxiliares se estimen oportunas.

13.- Vamos a solucionar el problema de las torres de Hanoi. Este problema consiste en tres varillas verticales clavadas en el suelo, de manera que en una de ellas se han ensartado una serie de rosquillas todas de diferente tamaño, de manera que las más grandes están abajo, y las de menor diámetro están arriba.

El problema consiste en pasar todas las rosquillas a la 3ª varilla (usando la segunda como almacenamiento temporal), de manera que nunca podemos situar una rosquilla grande sobre una

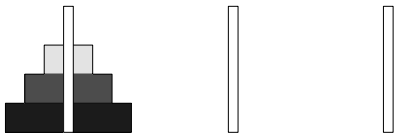
más pequeña.

A este problema, se le pasa un parámetro que es el número de rosquillas que hay que trasladar, y debe retornar la secuencia de operaciones de traslación necesarias para resolver el problema. Ej.:
Resolver(3) daría:
Pasar(**a**, **c**, Pasar(**b**, **c**, Pasar(**b**, **a**, Pasar(**a**, **c**, Pasar(**c**, **b**, Pasar(**a**, **b**, Pasar(**a**,**c**, Inicio)))))))).

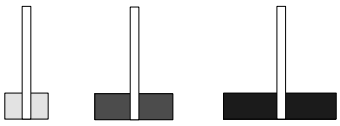
Podemos considerar que estamos creando en TAD Hanoi, de manera que tenemos el constructor: Solución_Parcial: Varilla x Varilla x Varilla x N x Hanoi --> Hanoi, donde la primera varilla representa la inicial, la seguna la intermediaria, y la tercera la final en la que deben acabar las rosquillas..

Se tendría el generador Inicio, que representaría la posición inicial de las rosquillas, y el generador Pasar: N_3×N_3×Hanoi ⟶ Hanoi.

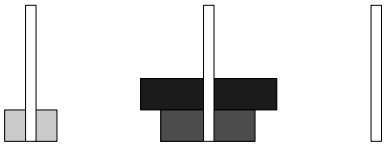
La solución al problema de pasar N elementos de la varilla **a** a la **c** usando como intermediaria la **b**, consiste en **1)** resolver el problema de N-1 elementos de la varilla **a** a la **b** usando como intermediaria la **c**, **2)** pasar el elemento N de la varilla **a** a la **c**, y **3)** resolver el problema de N-1 elementos de la varilla **b** a la **c** usando como intermediaria la **a**.



Estado inicial.



Estado correcto.



Estado incorrecto.