TEMA 1 Introducción a los TADs

ESTRUCTURAS DE DATOS

Objetivos

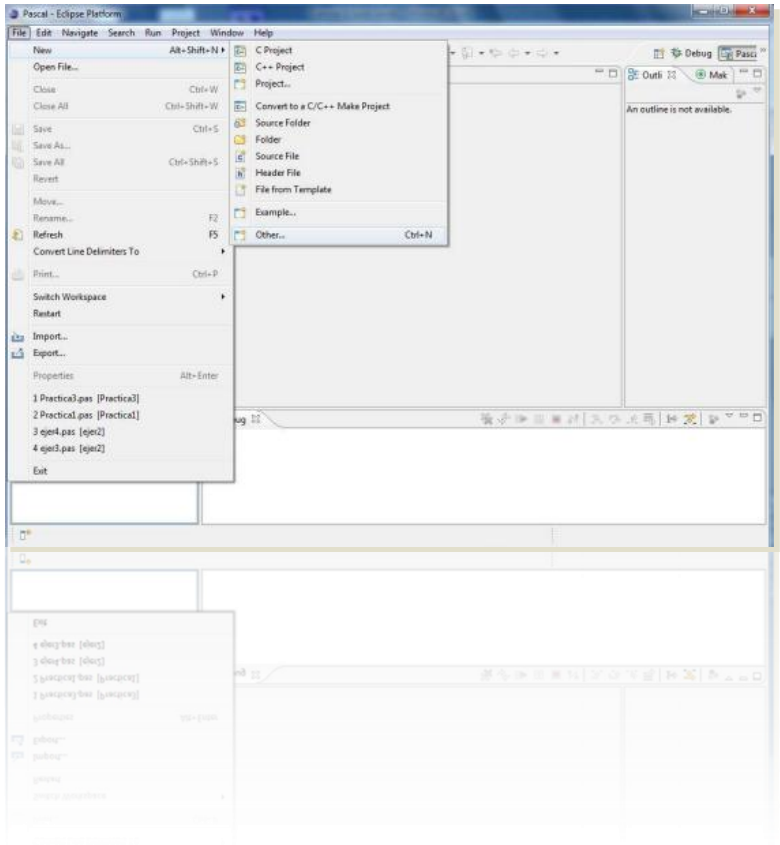
- Preliminares, eficiencia y corrección
  - Análisis de complejidad algorítmica en notación O()
- Abstracción de datos
- Presentar los Tipos Abstractos de Datos (TAD's)
- Presentar la especificación algebraica de TAD's

Contenidos

- 1.1 Preliminares
  - Normas de estilo
  - Conceptos aprendidos
  - Paradigmas y lenguajes de programación
  - Eficiencia y corrección
- 1.2 Abstracción de datos y TAD’s
  - Diseño basado en abstracciones: abstracción procedimental y de datos
  - Definición de TAD’s y realización de TAD’s en Pascal
  - Especificación algebraica de TAD’s

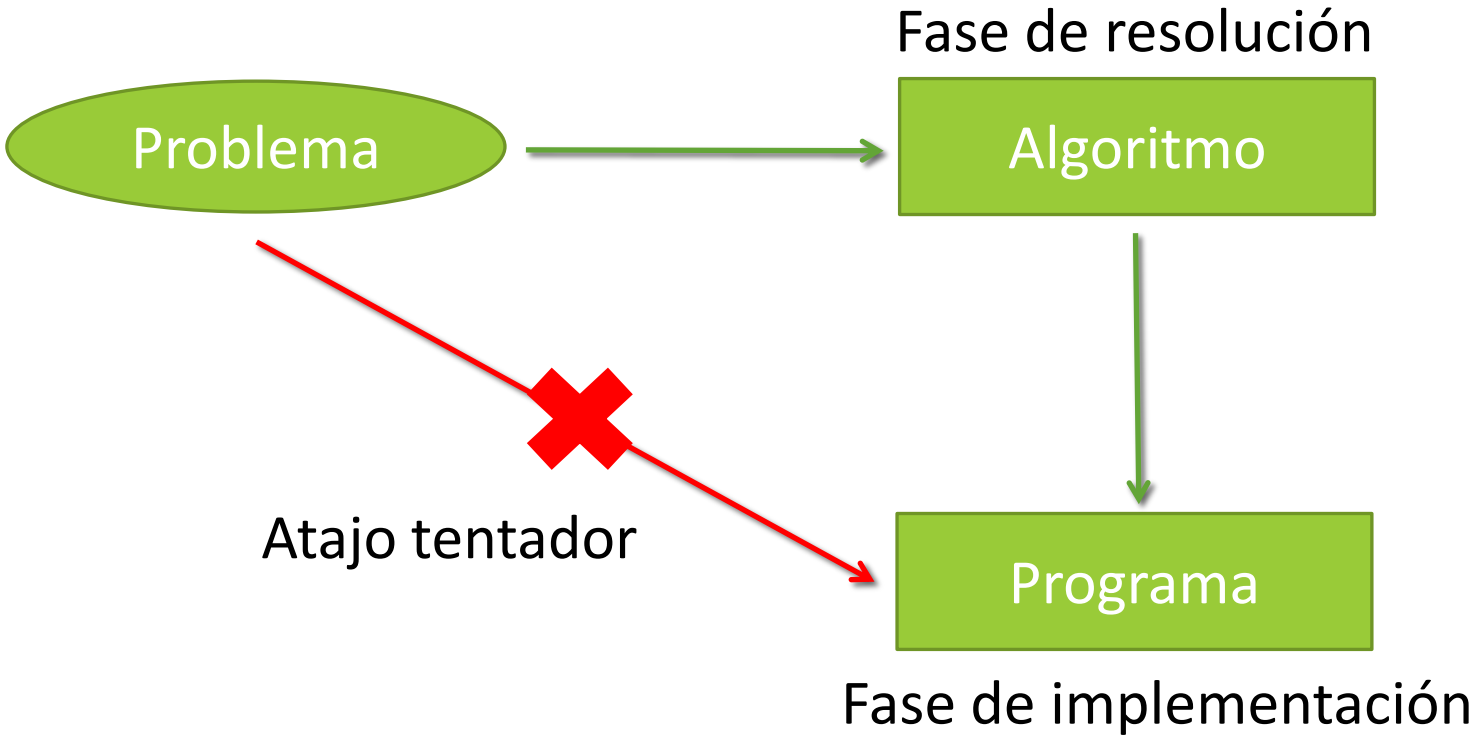
1.1 Preliminares

- ¿Qué debemos saber?
  - El lenguaje Pascal
    - Entorno EclipseGavab
      - https://github.com/codeurjc/eclipse-gavab
      - https://bintray.com/sidelab-urjc/EclipseGavab
    - Soltura con la sintaxis e instrucciones de control
    - Especialmente Arrays, Registros y Punteros
- Ciclo de Vida del Software
  - Análisis: Qué se hace
  - Diseño: Cómo se hace
  - Codificación: Se hace
  - Pruebas: Se prueba y se corrige
  - (Mantenimiento: Se corrige y amplía)



1.1 Preliminares

- Conceptos aprendidos



1.1 Preliminares: normas de estilo

- Eliminar varias instrucciones en una misma línea
- Tabular adecuadamente el anidamiento que generen algunas sentencias:
  - Evitar:

```
IF precio>MAXIMO THEN writeln('Precio abusivo');
```

- Salvo contadores, identificadores mnemotécnicos que describan cometido o lo que representan (subprogramas y variables).
- Palabras reservadas en mayúsculas
  - **WHILE, FOR, RECORD,...**

1.1 Preliminares: normas de estilo

- Identificadores: descriptivos y minúsculas
  - Identificadores con varias palabras, unidas por '_', o con el primer carácter de la palabra sufija en mayúscula, o ambos
    - Ej.: **`nombre_archivo`**, **`nombreArchivo`**, **`nombre_Archivo`**, …
  - Constantes en mayúsculas
    - Ej.: **`IVA`**, **`PI`**, **`NUMERO_E`**, …
  - Procedimientos: Empezando por letra mayúscula.
    - Ej.: **`BusquedaBinaria`**, **`Apilar`**, **`PilaVacia`**, …
  - Tipos: Empezando por “Tipo” o “T”
    - Ej.: **`TipoPila`**, **`TPila`**, …

1.1 Preliminares: normas de estilo

- Módulos y Ficheros:
  - Nombres de programas y módulos (unidades) deben coincidir con los nombres de los ficheros que los contienen.
  - Empezando por mayúsculas y resto minúsculas
  - Escribir una cabecera de identificación como la que se muestra a continuación:

```
{*****************************************************************
*                                                                *
* Módulo: Nombre                                                 *
* Fichero: ( ) Programa ( ) Espec. TAD ( ) Impl. TAD ( ) Otros   *
* Autor(es): Nombre(s)                                           *
* Fecha: Fecha de actualización                                  *
*                                                                *
* Descripción:                                                   *
* Breve descripción del módulo (párrafo corto)                   *
*                                                                *
*****************************************************************}
```

1.1 Preliminares: normas de estilo

- Uso extendido de subprogramas para tareas bien identificadas
- Adecuado uso de sentencias de repetición (especialmente bucles `FOR` y `WHILE`)
  - Esquema de búsqueda vs recorrido
- Evitar variables globales en subprogramas
- Uso adecuado de funciones
  - Devuelven un único valor

1.1 Preliminares: conceptos aprendidos

- Arrays en Pascal:
  - Un tipo de dato array se define en la sección de declaración de tipos `TYPE`

```
TYPE
  TipoCoordenada = ARRAY [1..3] OF real;
  TipoVector = ARRAY [1..3] OF real;
  TipoMatriz = ARRAY [1..3, 1..7] OF char;
  TipoViviendas = ARRAY [1..3, 1..3, 'A'..'E']OF boolean;

VAR
  origen: TipoCoordenada;
  desplazamiento: TipoVector;
```

1.1 Preliminares: conceptos aprendidos

- Arrays en Pascal (cont.):
  - Los arrays son **estructuras de acceso directo**, ya que permiten almacenar y recuperar directamente los datos, especificando su posición dentro de la estructura.
  - Los arrays son **estructuras de datos homogéneas**: sus elementos son **todos** del **mismo tipo**.
  - El **tamaño** de un array se establece de forma **fija**, en un programa, cuando se define una variable de este tipo.
  - **Cuidado** con **paso** de arrays como parámetros de **tipo anónimo** a subprogramas.

1.1 Preliminares: conceptos aprendidos

- Registros en Pascal:
  - Tipo de datos **estructurado** que permite almacenar datos **heterogéneos** (de distintos tipos)

```
TYPE
  TNombreReg = RECORD
    idCampo1 : idTipo1;
    idCampo2 : idTipo2;
             ...
    idCampon : idTipon
  END; {Fin de TNombreReg}
```

1.1 Preliminares: conceptos aprendidos

- Registros en Pascal (cont.):
  - Las funciones no pueden devolver un registro.
  - Para acceder a un campo se usa el operador punto (.):

```
NombreVariableRegistro.NombreCampo
```

  - También se puede acceder a los campos de un registro “abriendo” el registro con la sentencia `WITH`
    - Absolutamente no recomendado

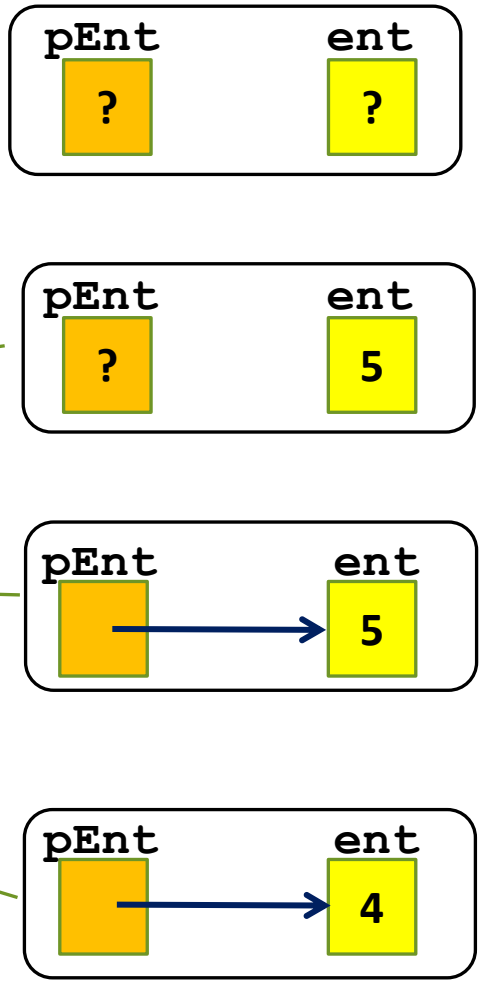
1.1 Preliminares: conceptos aprendidos

- Punteros en Pascal:
  - La gestión de memoria dinámica se realiza a través de los denominados **punteros a memoria**
  - Una variable puntero sirve para indicar cuál es la posición de memoria (potencialmente de otra variable)
  - Posee mecanismos para reservar y liberar posiciones de memoria en tiempo de ejecución
    - Generando variables dinámicas
  - A través del puntero se puede acceder al valor del dato que apunta

1.1 Preliminares: conceptos aprendidos

- Punteros en Pascal:

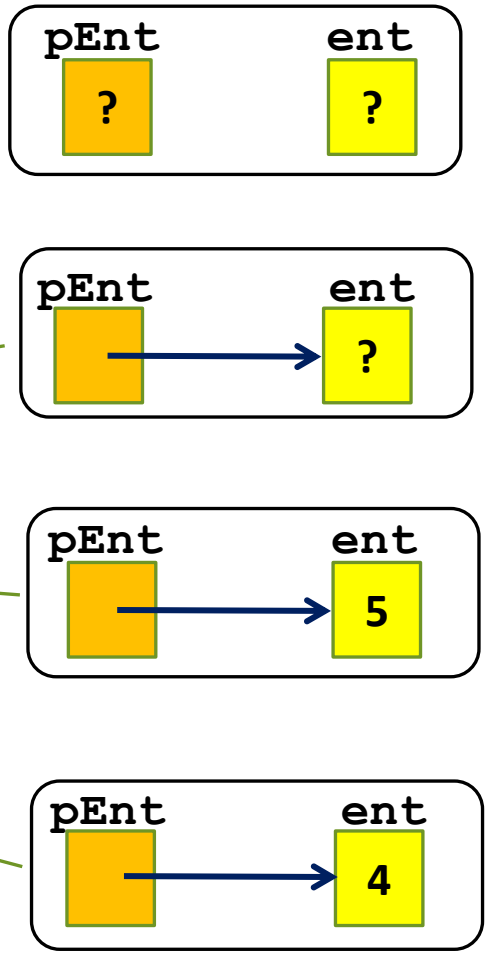
```
PROGRAM prueba1;
TYPE
  tPEntero = ^integer;
VAR
  pEnt: tPEntero;
  ent: integer;
BEGIN
  ent := 5;
  writeln(ent);
  pEnt := @ent;
  pEnt^ := 4;
  writeln(ent);
END.
```



1.1 Preliminares: conceptos aprendidos

- Punteros en Pascal:

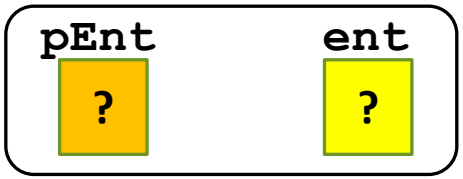
```
PROGRAM prueba1;
TYPE
  tPEntero = ^integer;
VAR
  pEnt: tPEntero;
  ent: integer;
BEGIN
  pEnt := @ent;
  ent := 5;
  writeln(ent);
  pEnt^ := 4;
  writeln(ent);
END.
```

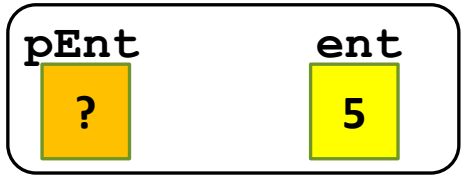


1.1 Preliminares: conceptos aprendidos

- Punteros en Pascal:

```
PROGRAM prueba1;
TYPE
  tPEntero = ^integer;
VAR
  pEnt: tPEntero;
  ent: integer;
BEGIN
  ent := 5;
  writeln(ent);
  pEnt^ := 4;
  pEnt := @ent;
  writeln(ent);
END.
```





Se produce un error! pEnt apunta a una posición indefinida de memoria y no reservada, por lo tanto, a través de él no se puede modificar el valor

1.1 Preliminares: conceptos aprendidos

- Punteros en Pascal:
  - Para utilizar una variable puntero no siempre hay que reservar memoria con él
    - Por ejemplo para recorrer una lista ya creada
  - Solo en ocasiones donde generamos datos en tiempo de ejecución utilizamos el procedimiento **`new`**
  - Si no necesitamos un área de memoria reservada anteriormente la debemos liberar con el método **`dispose`**
  - Para apuntar a “nada” utilizamos la constante **`NIL`**

1.1 Preliminares: conceptos aprendidos

- Punteros en Pascal:

```
TYPE
  tLista = ^tNodoLista;
  tNodoLista = RECORD
    informacion: Integer;
    sig: tLista  {Estructura Recursiva}
  END;

VAR
lista: tLista;
```

1.1 Preliminares: Evolución de los LPs

- Los motores que impulsan el desarrollo de los lenguajes de programación son:
  - **Abstracción**
  - **Modularidad**
  - **Encapsulación**
  - **Jerarquía**

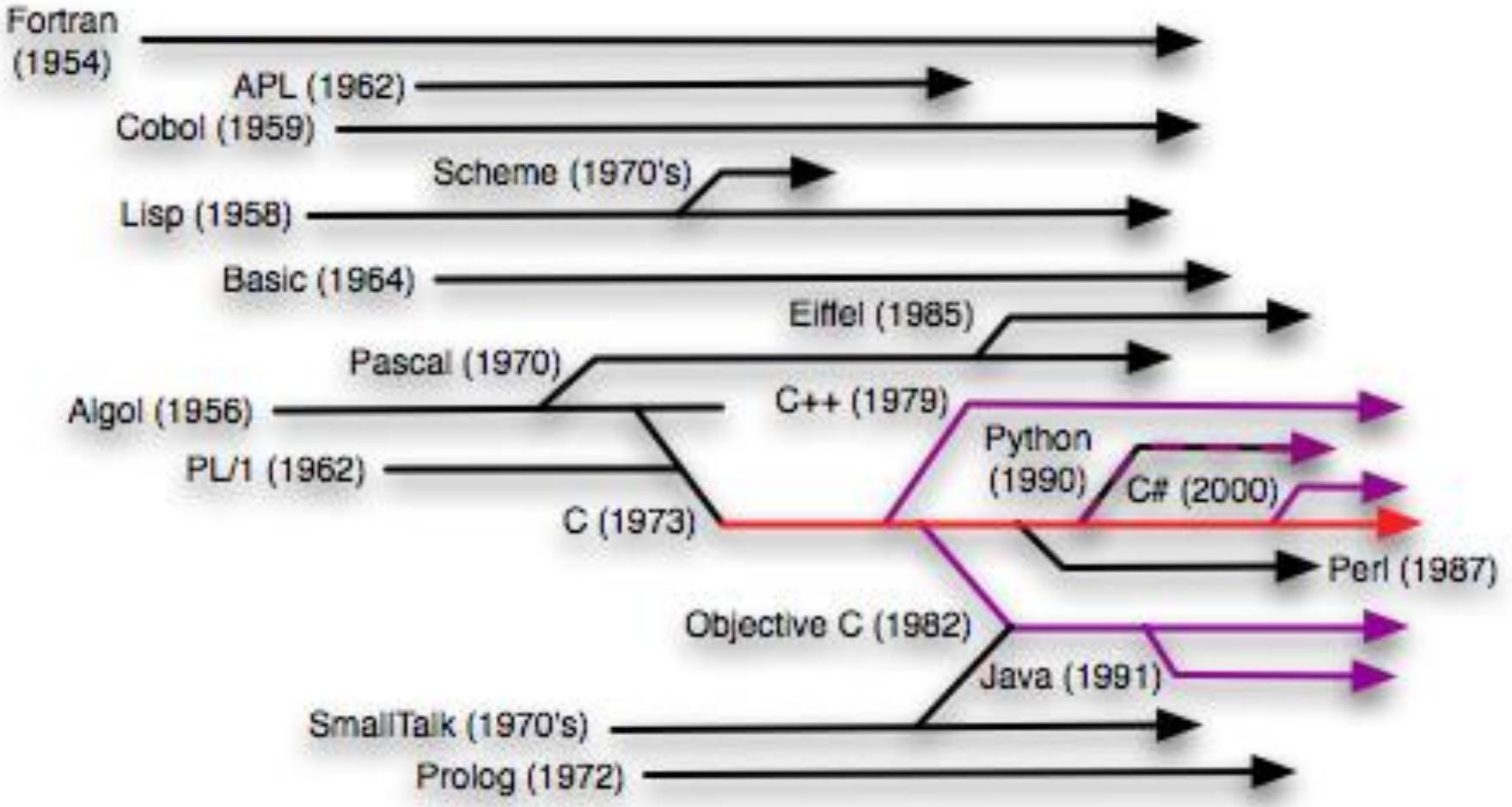
1.1 Preliminares: Evolución de los LPs

- **Abstracción**: Proceso mental por el que el ser humano extrae las características esenciales de algo, e ignora los detalles superfluos.
- **Modularización**: Proceso de descomposición de un sistema en un conjunto de elementos poco acoplados (independientes) y cohesivos
- **Encapsulación**: Proceso por el que se ocultan los detalles de las características de una abstracción
- **Jerarquía**: Proceso de estructuración por el que se organizan un conjunto de elementos en diferentes niveles atendiendo a unos criterios determinados

1.1 Preliminares: Evolución de los LPs

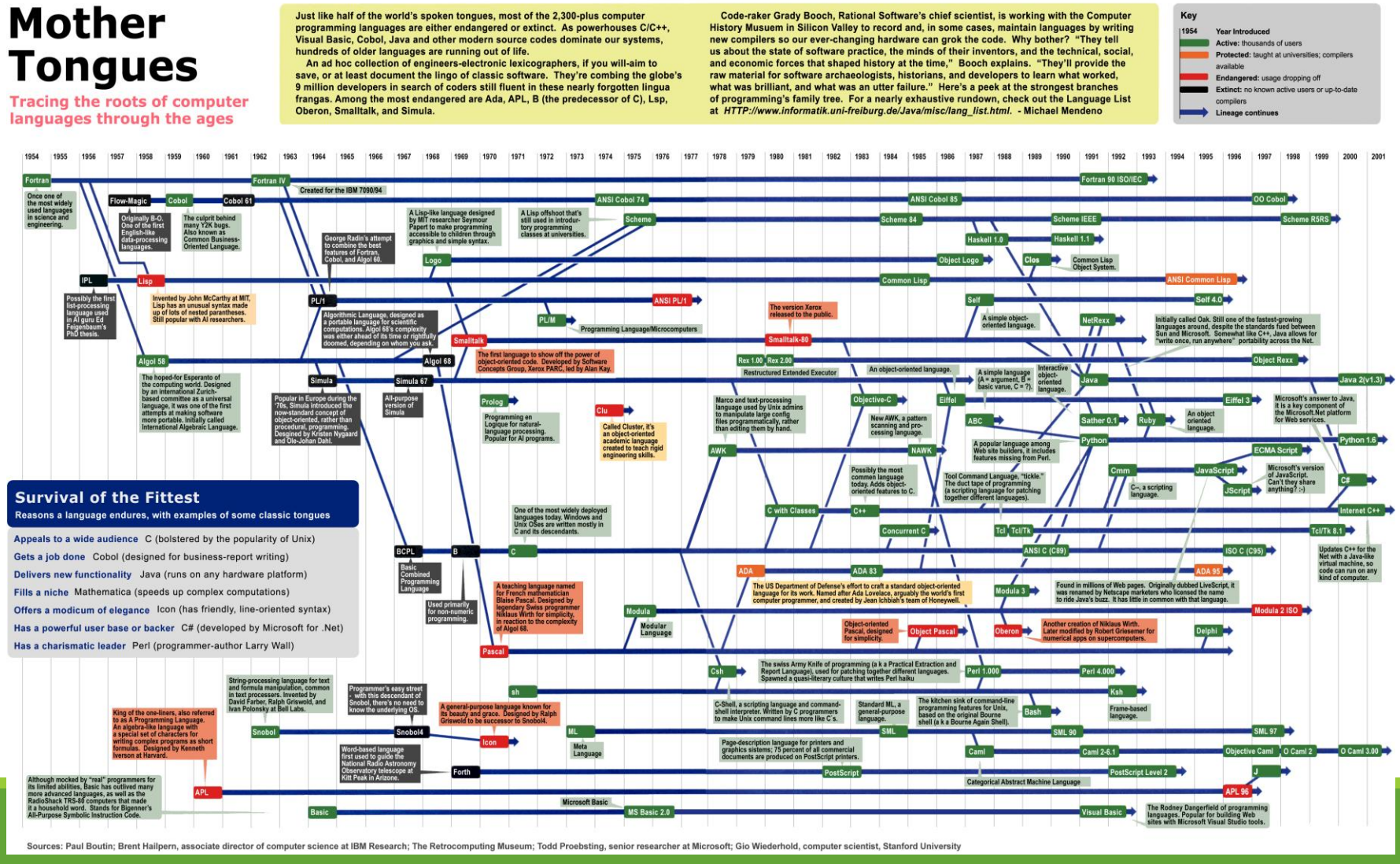
- Código máquina
- Lenguaje Ensamblador
- Lenguaje de alto nivel
  - Programación Estructurada
  - Programación Modular
  - Programación con TAD's (Tipos Abstractos de Datos)
  - Programación Orientada a Objetos

1.1 Preliminares: Evolución de los LPs

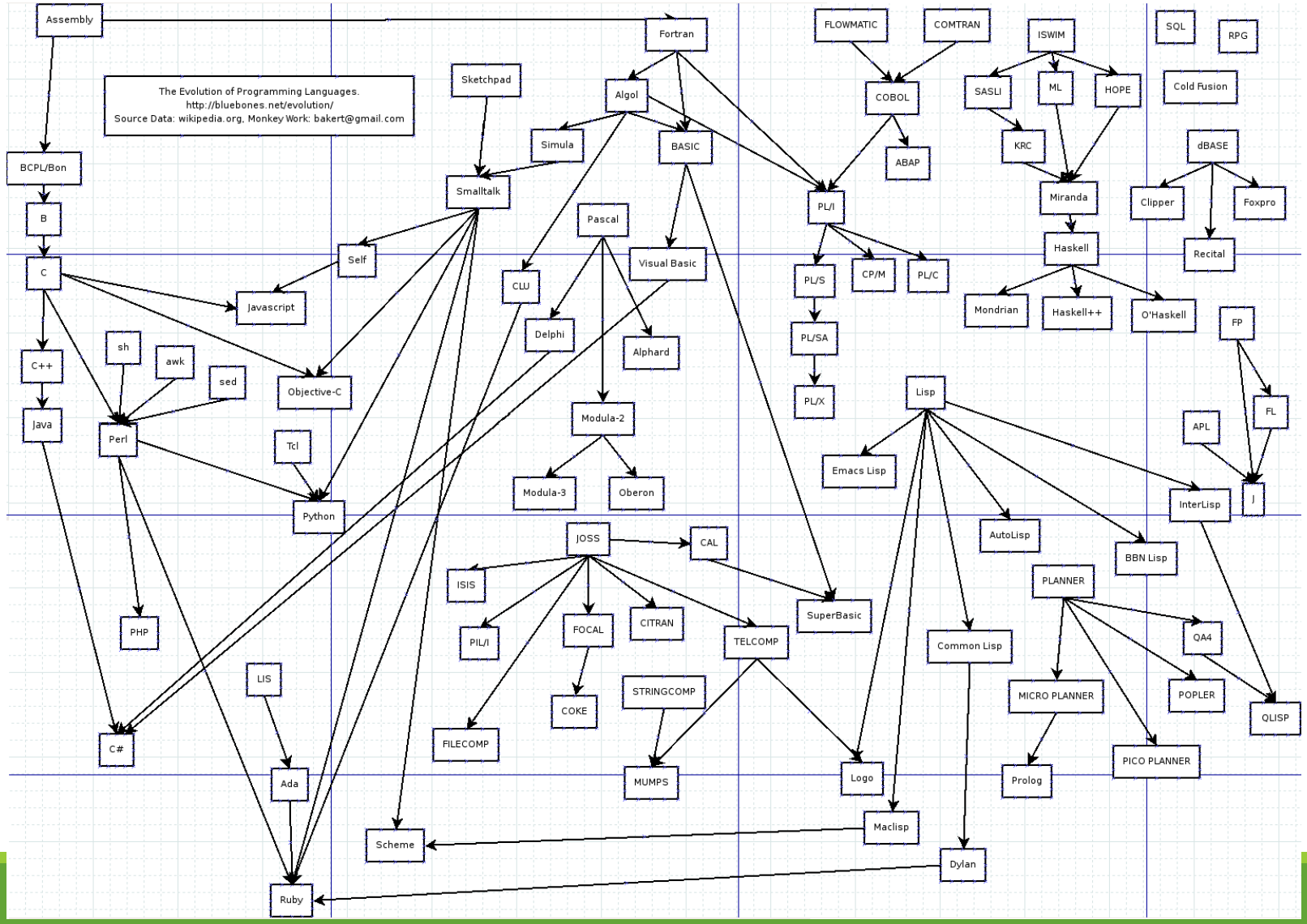


1.1 Preliminares: Evolución de los LPs

http://www.digibarn.com/collections/posters/tongues/ComputerLanguagesChart.png



1.1 Preliminares: Evolución de los LPs



1.1 Preliminares: Paradigmas de programación

- Un paradigma de programación se define como una colección de patrones conceptuales que moldean la forma de razonar sobre problemas, de formular algoritmos y de estructurar programas
- Paradigmas:
  - Programación imperativa
  - Programación funcional
  - Programación lógica
  - Programación orientada a objetos

1.1 Preliminares: Programación imperativa

- Basada en comandos que actualizan variables que están en almacenamiento
- Permite cierto nivel de abstracción: variables, expresiones, instrucciones
- Para programar:
  - Declarar las variables necesarias
  - Diseñar una secuencia (algoritmo) adecuada de instrucciones (asignaciones)

1.1 Preliminares: Programación funcional

- Tiene su base en el concepto de función matemática:

  f: dominio --> rango

- Para programar:
  - Se construyen funciones sencillas
  - Se construyen funciones más complejas a partir de las sencillas
  - Se evalúan las funciones sobre los datos de entrada

1.1 Preliminares: Programación lógica

- Tiene su base en cálculo de predicados de primer orden (hechos y reglas)
- Para programar:
  - Se definen hechos (o predicados básicos)
  - Se diseñan implicaciones para definir predicados complejos
  - Se determina la veracidad de los predicados para individuos concretos

1.1 Preliminares: Programación orientada a objetos

- Basado en el concepto de abstracción de datos
- El programa se organiza alrededor de los datos
- Misma filosofía que TAD's (en clases) pero extendida mediante otros conceptos:
  - Herencia
  - Polimorfismo

1.1 Preliminares: Eficiencia y corrección

- Propiedades exigibles a los programas
  - **Corrección**: el programa debe funcionar
  - **Eficiencia**: Aprovechamiento adecuado de los recursos (tiempo y memoria)
  - **Claridad**: Debe estar documentado (legibilidad, comentarios, tabulado, nombres adecuados para identificadores,...)

1.1 Preliminares: Eficiencia y corrección

- Estudio de la eficiencia de un programa
  - Memoria necesaria para almacenar datos e instrucciones del programa
  - Tiempo que tarda en ejecutarse un programa
    - Contar el número de instrucciones que de cada tipo se ejecutan y multiplicarlo por el tiempo de ejecución de cada una de las diferentes instrucciones.
    - Otra manera es calcularlo en función del tamaño de los datos de entrada: Notación O(), notación $\theta()$ y notación $\Omega()$.

1.1 Preliminares: Eficiencia y corrección

- Notación asintótica O()
  - Se dice que $f(n) = O(g(n))$ (f de n es o grande de g de n) si y sólo si existen constantes positivas c y n_0 tales que $f(n) \leq cg(n)$ para todo $n \geq n_0$.
  - Esta definición es demasiado general puesto que para $n = O(n^2)$ y por tanto también $n = O(n^3)$
  - Por ello se suele tomar por cota superior la de menor orden.

1.1 Preliminares: Eficiencia y corrección

- Reglas notación O()
  - Asignaciones y expresiones simples: O(1)
  - Secuencia de instrucciones (regla de la suma)
    - Tiempos de ejecución de una secuencia de instrucciones es igual a la suma de sus tiempos de ejecución respectivos.
      - $T(I_1; I_2) = T(I_1) + T(I_2)$
      - $O(T(I_1; I_2)) = max\ (O(T(I_1)), O(T(I_2)))$ *regla del máximo*
  - Instrucciones de selección (IF..THEN..ELSE; CASE)
    - Utilizamos la regla del máximo

1.1 Preliminares: Eficiencia y corrección

- Reglas notación O() (cont.)
  - Instrucciones de repetición (FOR; WHILE; REPEAT...UNTIL)
    - Si el coste del cuerpo no depende de la iteración:

      $O(f_{iter}(n) \times f_{cuerpo}(n))$
    - Si el coste del cuerpo depende de la iteración:

$$T_I(n) \in O\left(\sum_{i=1}^{f_{iter}(n)} f_{cuerpo}(i)\right)$$

1.1 Preliminares: Eficiencia y corrección

- Reglas notación O() (cont.)
  - Para programas recursivos
    1. Se plantea la ecuación de recurrencia para T(n), que incluye el caso base y el general
    2. Se resuelve la ecuación de recurrencia
       - ↓ Una manera de hacerlo es mediante la expansión de recurrencias hasta dar con un término general

1.1 Preliminares: Eficiencia y corrección

- Otras notaciones
  - También podemos estudiar la cota inferior (notación Omega) y un caso medio (notación Theta)
  - Se dice que $f(n) = \Omega(g(n))$ (*f* de *n* es *omega* de *g* de *n*) si y sólo si existen constantes positivas *c* y n_0 tales que $f(n) \geq cg(n)$ para todo $n \geq n_0$.
  - Se dice que $f(n) = \Theta(g(n))$ (*f* de *n* es *theta* de *g* de *n*) si y sólo si existen constantes positivas c_1, c_2 y n_0 tales que $c_1 g(n) \leq f(n) \leq c_2\, g(n)$ para todo $n \geq n_0$.

1.2 Abstracción de datos y TAD's

- Abstracción:
  - Proceso por el que ciertas propiedades o características de objetos (o acciones) se ignoran, al ser periféricas e irrelevantes.
- Es la clave para resolver problemas e implica:
  - Identificar elementos importantes del problema
  - Nombrarlos
  - Plantear y resolver el problema "abstracto" y adaptar la solución al problema real.

Abstracción de datos y TAD's

- La abstracción se usa para descomponer un problema complejo en subproblemas más pequeños y manejables.
- Evolución de los lenguajes ejemplo de la potencia de la abstracción
  - Lenguajes de alto nivel permitieron trabajar de modo independiente de las máquinas concretas
  - La idea de procedimiento con parámetros permite dar un nombre a un conjunto de instrucciones y activarlo con una sola instrucción

Abstracción de datos y TAD's

- La historia de la programación es un camino hacia un grado creciente de abstracción.
  - Claro ejemplo es la programación orientada a objetos

Diseño basado en abstracciones

- Programas de cierto tamaño y complejidad pueden ser difíciles de manejar
- Modularización y diseño descendente (o “top-down”) aportan una solución y está asociado al proceso de refinamiento
- Un programa lo dividimos en tareas, que a su vez se dividen en subtareas,...

Esfuerzo(p1+p2) > Esfuerzo(p1) + Esfuerzo(p2)

Diseño basado en abstracciones

- Esta división lógica permite agrupar subtareas (subprogramas) en módulos (unidades en Pascal, paquetes en ADA,...)
- El programa se empieza a llenar de “cajas negras”
- Según avanzamos en el refinamiento cada “caja negra” da lugar a otras hasta llegar a sentencias en Pascal.
- Cada caja negra especifica QUÉ hace el subprograma pero no CÓMO.

Diseño basado en abstracciones

- Tipo de abstracciones:
  - Abstracción funcional o procedimental: usuario sólo necesita la especificación (**qué** hace) para utilizarlo, ignorando los detalles (**cómo** está hecho o cómo se hace)
  - Abstracción de datos:
    - Tipos de datos: representación irrelevante para el usuario
    - Tipos definidos por el programador: definición de valores cercanos al problema
    - Tipos abstractos de datos (TAD's): definición y representación de tipos de datos (valores y operaciones)
    - Clases de Objetos: Extensión de los TAD's para soportar herencia, polimorfismo.

Definición de TAD's

- Concepto propuesto hacia 1974 por John Guttag y otros investigadores del MIT.
- Un tipo abstracto de datos es una **colección de valores y de operaciones** que se definen mediante una especificación que es independiente de cualquier representación.
- Lo llamaremos abreviadamente TAD, aunque es común verlo como ADT (del inglés) o TDA

Definición de TAD's

- Un tipo de dato: colección de objetos y conjunto de operaciones que actúan sobre ellos.
- Por ejemplo el tipo INTEGER
  - Objetos {0, +1, -1, +2, -2,..., maxint, minint}
  - Operaciones definidas para esos objetos:
    - Operadores +, -, *, DIV
    - Igualdad, desigualdad
    - Asignación a una variable

Definición de TAD's

- Además de saber esto podríamos querer saber cómo se representan en la máquina.
  - En la mayoría de ordenadores un CHAR se representa mediante una cadena de bits ocupando 1 byte de memoria
  - Análogamente un entero puede representarse por 2 bytes por lo que maxint = $2^{15} - 1 = 32767$

Definición de TAD's

- Conocer estos detalles de representación puede ser útil pero peligroso
  - Ventaja: Podemos construir algoritmos que hagan uso de ello
  - Desventaja: cualquier cambio en la representación conlleva hacer cambios en los algoritmos
- Se ha demostrado que en muchas ocasiones es preferible ocultar muchos aspectos de bajo nivel de los datos
  - En ese caso el usuario queda restringido a manipular los objetos mediante las funciones proporcionadas

Definición de TAD's

- Se definen con la intención de especificar las propiedades lógicas de un tipo de dato
- La definición de un TAD implica la definición de dos partes principalmente:
  - Interfaz público
  - Representación privada o implementación
- Los objetivos son:
  - Privacidad de la representación
  - Protección
  - Facilidad de uso

Definición de TAD's

- Algunos lenguajes tienen mecanismos explícitos para soportar la distinción entre la especificación y la implementación
  - De esta manera podemos separarlo físicamente
    - Ada (package): ".ads" y ".adb"
    - C++ (class): ".h" y ".cpp"
    - Turbo Pascal (unit): un sólo archivo: ".tpu"

Definición de TAD's

- En el primer nivel de diseño se debe ver el tipo abstracto de datos como una "caja negra"
- No debemos preocuparnos por el cómo implementar las operaciones del tipo de dato sino por el qué implementar
  - Existen frigoríficos en el mercado que dependiendo de dos mecanismos son capaces de suministrar agua o hielo. No nos importa cómo lo hace sino lo que hace
  - Un coche según el punto de vista de un usuario no es lo mismo que desde el punto de vista de un mecánico o de un fabricante. Nosotros adoptamos el rol del usuario en esta primera fase de diseño.

Definición de TAD's

- Por tanto lo primero será especificar lo mejor posible lo que se quiera hacer
- Esta especificación se puede hacer desde un enfoque relativamente formal o más práctico
- Dentro de los enfoques formales tenemos varias posibilidades entre las que destaca la especificación axiomática o algebraica.

Realización de TAD's en Pascal

- Para la implementación de tipos abstractos de datos en Turbo Pascal utilizaremos las unidades.
- Una unidad (`UNIT`) contiene una parte de interfaz y otra de implementación.
  - Interfaz:
    - Pública (visible desde el programa externo)
  - Implementación:
    - Privada (no visible desde el programa externo)

Realización de TAD's en Pascal

- La plantilla de generación de unidades:

```
UNIT NombreTAD;

[ PARAMETROS FORMALES:

{lista de tipos que parametrizan una especificación} ]

INTERFACE

[USES lista_de_unidades]

 {def. de constantes, tipos de datos y variables públicas}

 {cabeceras de las funciones y procedimientos públicos (con especificación del tipo
 de operación)}

IMPLEMENTATION

[USES lista_de_unidades]

 {def. de constantes, tipos de datos y variables privadas}

 {def. de las funciones y procedimientos públicos y privados}

[begin

 {sentencias de inicialización opcionales}]

END. {unit}
```

Realización de TAD's en Pascal

- Como se puede observar, las unidades fuerzan la abstracción procedimental.
- Podemos compilar la unidad como si fuese un programa
- Para usar la unidad desde un programa simplemente añadimos la cláusula:

```
USES nombre_de_la_unidad
```

- Con ello tenemos disponibilidad de hacer llamadas a los subprogramas declarados en la parte pública de la unidad (interfaz)

Realización de TAD's en Pascal

- Debemos definir TODAS las operaciones que queramos que sean válidas para el manejo de nuestro TAD's
- Primeramente lo haremos atendiendo a QUÉ se puede hacer con el TAD
- Después de haberlo declarado en la parte de interfaz prestaremos atención al cómo lo hacemos (en la sección de implementación)

Realización de TAD's en Pascal

- Imaginemos un caso sencillo de TAD: Números complejos
- El TAD en sí consta de la definición del tipo de dato y de las operaciones que veamos oportunas para el manejo del mismo
- Por ejemplo y de forma sencilla:
  1. Crear uno a partir de 2 reales (CrearComplejo)
  2. Sumar y restar
  3. Devolver la parte real dado un complejo
  4. Devolver la parte imaginaria dado un complejo

Realización de TAD's en Pascal

- El fichero lo guardamos con el nombre dado a la unidad (extensión *.PAS*)
- Lo compilamos para generar un archivo con extensión *.TPU* (*Turbo Pascal Unit*)
- Esta unidad está ahora accesible a cualquier programa que la utilice mediante `USES`

Realización de TAD's en Pascal

- En el programa se declara una variable de un tipo definido dentro de una unidad
- Si no usáramos `USES` no tendríamos posibilidad de hacerlo sin error (`Unknown Identifier`)
- Ahora podemos usar esa variable desde nuestro programa como cualquier otra.

Realización de TAD's en Pascal

- En el programa principal podríamos hacer algo como:

```
BEGIN
  complejo.re := 3;
  complejo.im := 5;
END.
```

- Podemos inicializar desde nuestro programa la variable declarada (no muestra error).
- Lo podemos hacer porque conocemos cómo está representado este tipo de dato

Realización de TAD's en Pascal

- Pero conocer cómo está representado el tipo de dato no es bueno
- Precisamente por eso debemos crear operaciones de construcción dentro de la unidad
- Como norma general, el acceso directo a los datos del TAD será evitado, y en este curso de ED está absolutamente prohibido por romper la filosofía del encapsulamiento y ocultación de la información.
- Lo único que necesitamos saber como usuarios son los parámetros que hay que pasar a las operaciones del TAD.

Realización de TAD's en Pascal

- Por tanto podríamos utilizar una operación constructora de la siguiente manera (siempre que estuviera definida en el TAD):

```
BEGIN
  CrearComplejo(3.5,5.1,complejo);
END.
```

- Así no necesitamos saber si el tipo complejo está definido como un registro o como un array de 2 componentes,…
- Tan sólo se hace una llamada a un procedimiento que vendrá especificado en la documentación del TAD

Realización de TAD's en Pascal

- Análogamente podemos escribir:

```
VAR
  comp1, comp2: TipoComplejo;

BEGIN
  CrearComplejo(2,3,comp1);
  CrearComplejo(3,5,comp2);
  Sumar(comp1, comp2);
  writeln(DevolverReal(comp1));
END.
```

- Dado el TAD nosotros lo utilizamos como una "caja negra" olvidándonos de cómo está hecho y preguntándonos qué queremos hacer

Especificación algebraica de TAD's

- Volviendo al ejemplo de números complejos, ahora podemos introducir más operaciones útiles para el manejo de números complejos
- ¿Qué operaciones necesitamos para el manejo de números complejos?
  1. Crear uno a partir de 2 reales
  2. Calcular el módulo
  3. Calcular el conjugado
  4. Sumar, restar, multiplicar y dividir 2 complejos
  5. Devolver la parte real dado un complejo
  6. Devolver la parte imaginaria dado un complejo

Especificación algebraica de TAD's

- Se puede observar que hay operaciones que son capaces de construir variables del tipo pedido y otras que no
- De manera general decimos que:
  - Las operaciones que devuelven el tipo a definir son operaciones constructoras
  - Las operaciones que devuelven otro tipo de dato son operaciones observadoras

Especificación algebraica de TAD's

- Dentro de las constructoras distinguimos entre:
  - Generadora: sólo con ellas es posible generar cualquier valor del tipo
  - No generadora o modificadoras: las que construyen con datos del mismo tipo que el tipo destino
- Análogamente entre las observadoras distinguimos:
  - Selectoras: las que seleccionan o acceden a partes integrantes del tipo
  - No selectoras: las que operan con el TAD para extraer otras propiedades.

Especificación algebraica de TAD's

- En este sentido podemos clasificar las operaciones anteriores para los números complejos como:
  1. Crear complejo a partir de 2 reales: **Constructora generadora**
  2. Calcular el conjugado: **Constructora no generadora**
  3. Sumar, restar, multiplicar y dividir 2 complejos: **Constructora no generadora**
  4. Devolver la parte real dado un complejo: **Observadora selectora**
  5. Devolver la parte imaginaria dado un complejo: **Observadora selectora**
  6. Calcular el módulo: **Observadora no selectora**

Especificación algebraica de TAD's

- ¿Por qué esta clasificación?

1. Crear Complejo a partir de 2 reales: **Constructora generadora**
   - Matemáticamente podemos expresar la operación CrearComplejo como:

     CrearComplejo: Real x Real → Complejo

   - Partimos de dos reales para crear de ellos un tipo Complejo: **Generamos**

Especificación algebraica de TAD's

2. Calcular el conjugado: **Constructora no generadora**

Conjugado: Complejo → Complejo

- Partimos de un Complejo para construir otro: **Construimos pero NO Generamos**

Especificación algebraica de TAD's

3. Sumar, restar, multiplicar, dividir: **Constructora no generadora**

Sumar: Complejo x Complejo → Complejo
Restar: Complejo x Complejo → Complejo
Multiplicar: Complejo x Complejo → Complejo
Dividir: Complejo x Complejo → Complejo

- Partimos de dos Complejos para construir otro: **Construimos pero NO Generamos**

Especificación algebraica de TAD's

4. y 5. Real, Imaginaria: **Observadora selectora**

Real: Complejo → Real

Imaginaria: Complejo → Real

- Partimos de un Complejo para observar algún atributo o miembro del tipo Complejo (Observamos parte integrante): **Observadora Selectora**

Especificación algebraica de TAD's

6. Módulo: **Observadora no selectora**

Módulo: Complejo → Real

- Partimos de un Complejo para extraer alguna propiedad del tipo Complejo: **Observamos pero NO Seleccionamos**

Especificación algebraica de TAD's

- Para algunas operaciones se deben especificar requisitos a cumplir
  - Por ejemplo, para dividir dos números complejos el denominador debe ser distinto de 0
  - El denominador de un cociente de números complejos es (tanto para la parte real como para la imaginaria): $(re_2^2 + im_2^2)$
  - Por tanto sólo se podrá definir la operación dividir cuando $(re_2 \neq 0)$ ó $(im_2 \neq 0)$

Especificación algebraica de TAD's

- Puesto que las operaciones pueden ser ambiguas debemos acotar su forma de actuar mediante una especificación adecuada.
- Además, el significado de un TAD no debe depender de su realización concreta (lenguaje, representación del tipo,...) sino de la especificación que defina su comportamiento.

Especificación algebraica de TAD's

- La especificación de una abstracción de datos incluye
  - Descripción de los valores que puede tomar
  - Descripción de las operaciones realizables sobre él
- Una forma sencilla (en lenguaje natural) de especificar una abstracción de datos es:
  - Tipo de dato: nombre
  - Valores: descripción de los valores posibles
  - Operaciones: especificación de cada operación

Especificación algebraica de TAD's

- Esta especificación se puede formalizar recibiendo el nombre de especificación algebraica
- Con ello se consigue:
  - Una especificación independiente de la implementación
  - Una interpretación única por parte de los usuarios
  - La posibilidad de verificar formalmente los programas de los usuarios del tipo
  - Deducción a partir de la especificación, de propiedades satisfechas por cualquier implementación válida de mismo.

Especificación algebraica de TAD's

- Utilizaremos la siguiente estructura
  - Tipo: Nombre del tipo abstracto de datos
  - Sintaxis: Forma de las operaciones
  - Semántica: Significado de las operaciones
- En la sintaxis se suministra la lista de funciones/procedimientos de la abstracción, así como los tipos de los argumentos y resultado
  - NombreOperación: TipoArgumentos → TipoResultado
- En la semántica se indica el comportamiento de las funciones/procedimientos definidos sobre la abstracción.
  - NombreOperacion(valores particulares) = expresión

Especificación algebraica de TAD's

- Para la semántica ha de tenerse en cuenta que:
  - Las reglas han de intentar aplicarse en el orden indicado
  - Ciertas operaciones no se definen; pueden considerarse axiomas o constructores de los valores del tipo
  - La expresión del resultado puede ser recursiva (conteniendo referencias a la misma función) o contener referencias a otras funciones del TAD
  - Las expresiones pueden contener referencias a otros tipos ya definidos o predefinidos (p. ej. Boolean)

Especificación algebraica de TAD's

- Esquema genérico para la especificación algebraica

```
TAD Nombre

[ PARAMETROS FORMALES:

<<lista de tipos que parametrizan una especificación>> ]

[ USA:

<<lista de tipos (abstractos) necesarios en la
  especificación>>]

OPERACIONES

...

[parcial] F_i: T_i -> R_i

...
```

(cont.)

Especificación algebraica de TAD's

```
VARIABLES:

...

Vi: <<variable del tipo Nombre o de los tipos de la lista PARAMETROS FORMALES
   o USA, que permiten escribir las ecuaciones>>

...

[ECUACIONES DE DEFINITUD:

...

Def(<<término bien definido para la operación parcial Fi>>)

...]

ECUACIONES:

...

<<expresan el comportamiento observable del TAD. El formato de cada ecuación
   es:

operación_no_generadora(operación_generadora(variables))=expresión >>

...

FTAD
```