RESUMEN – CONCEPTOS Y PARADIGMAS

Teoría 1: Introducción y evaluación de lenguajes

Criterios para evaluar los lenguajes de programación:

- **Simplicidad y legibilidad:** producir programas fáciles de leer y escribir. Fácil a la hora de aprenderlo y enseñarlo.
- **Claridad en los bindings (ligadura)**: la ligadura debe ser clara y los elementos pueden ligarse a sus atributos en diferentes momentos (definición, implementación, etc.)
- **Confiabilidad:** relacionada con la seguridad -> chequeo de tipos, manejo de excepciones.
- **Soporte:** lenguaje accesible para cualquier uso. Facilidad de implementación en diferentes plataformas. Existencia de medios de aprendizaje como tutoriales, cursos, etc.
- **Abstracción:** definir y usar estructuras/operaciones para ignorar detalles.
- **Ortogonalidad:** conjunto pequeño de constructores primitivos puede ser combinado a la hora de construir estructuras de control y datos.
- **Eficiencia:** tiempo y espacio. Esfuerzo humano. Optimizable.

Teoría 2: Sintaxis

Sintaxis: conjunto de reglas que definen como componer letras, dígitos y otros caracteres para formar los programas.

- **Características:** ayuda al programador a escribir programas correctos sintácticamente. Establece reglas para la comunicación entre el programador y el procesador. Legible, verificable, no ambigüa.
- **Elementos**
  - Alfabeto – Conjunto de caracteres
  - Identificadores
  - Operadores
  - Palabra clave (tienen un significado dentro de un contexto) – Palabra reservada (palabras claves que no pueden ser usadas como identificador)
  - Comentarios – Uso de blancos
- **Estructura sintáctica**
  - Vocabulario / Words: caracteres y palabras necesarias para construir expresiones, sentencias y programas.
  - Expresiones: funciones que devuelven un resultado. Bloques sintácticos básicos a partir de los cuales se construyen sentencias y programas.

    - Sentencias: Componente más importante. Hay simples, estructuradas y anidadas.
- **Reglas léxicas y sintácticas**
    - Léxicas: reglas para formar las words a partir de los caracteres. Ejemplo: símbolo de distinto, case sensitive.
    - Sintácticas: reglas para formar las expresiones y sentencias. Ejemplo: IF lleva/no lleva then.
- **Tipos de sintaxis**
    - Abstracta: hace referencia a la ESTRUCTURA.
    - Concreta: hace referencia a la PARTE LÉXICA.
    - Pragmática: hace referencia al USO PRÁCTICO.
- **Formas de definir la sintaxis**
    - Lenguaje natural
    - Gramática libre de contexto: BNF. Notación formal para describir sintaxis. Es un metalenguaje y usa metasímbolos. BNF Extendida -> EBNF.
    - Arboles sintácticos: dos maneras de construirlos -> método bottom-up | método top-down.
    - Diagramas sintácticos: Elementos -> Terminales, Flujo, Repetición, Selección, etc.

Teoría 3: Semántica

Semántica: describe el significado de los símbolos, palabras y frases de un lenguaje natural o informático.

- **Tipos de semántica**
    - Estática: relacionado con las formas válidas. El análisis para el chequeo puede hacerse en compilación. Para describirlas formalmente se usan las *gramáticas de atributos*.
        - Gramáticas de atributos: a los símbolos de la gramática se le asocia información denominada ATRIBUTOS. Los valores de los mismos se calculan mediante ECUACIONES/REGLAS SEMÁNTICAS. La forma de expresar la gramática se escribe en forma tabular.
    - Dinámica: describe el efecto de ejecutar diferentes construcciones en el lenguaje. Su efecto se describe durante la ejecución del programa. Programa sólo se ejecuta si son correctos para sintaxis + semántica estática.
- **Forma de describir la semántica**
    - No existen herramientas estándar. Soluciones formales:
        - Semántica axiomática: considera al programa como máquina de estados. Notación: cálculo de predicados. Se describe indicando cómo la ejecución del constructor provoca cambio de estado.

Predicado: describe valores de variables en ese estado.
Posee pre-condición y post-condición.

- Semántica denotacional: basado en teoría de funciones recursivas. Diferencia con la axiomática -> describe predicados a través de funciones. Se define correspondencia entre constructores sintácticos y sus significados.
- Semántica operacional: significado de un programa se describe mediante otro lenguaje de bajo nivel implementado sobre máquina abstracta. Es un método informal y el más utilizado en los libros de texto.

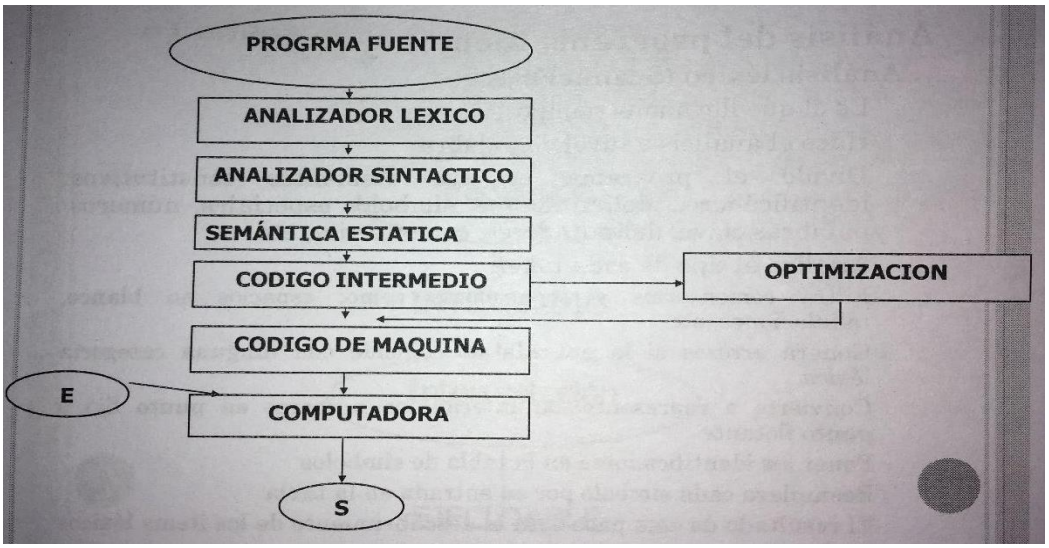
Procesamiento de un lenguaje

- **Traducción:** traducir lenguajes ensambladores a lenguaje de máquina.
  Alternativas:
  - **Interpretación:** el intérprete lee, analiza, decodifica y ejecuta una a una las sentencias de un programa escrito en lenguaje de programación. La interpretación es realizada llamando a subprogramas en la secuencia adecuada. Ejecuta repetidamente: obtener próxima sentencia, determina acción a ejecutar, ejecuta la acción.
  - **Compilación:** programas escritos en lenguaje de alto nivel se traduce a versión de lenguaje de máquina antes de ser ejecutados. Lleva varios pasos.

Diferencias entre compilador (traductor) e intérprete

	INTÉRPRETE	COMPILADOR
FORMA EN CÓMO EJECUTA	Ejecuta el programa de entrada directamente	Produce un programa equivalente en lenguaje objeto
FORMA EN QUÉ ORDEN EJECUTA	Sigue el orden lógico de ejecución	Sigue el orden físico de las sentencias
TIEMPO DE EJECUCIÓN	Por cada sentencia se realiza el proceso de decodificación para determinar las operaciones a ejecutar y sus operandos	No repetir lazos, se decodifica una sola vez
EFICIENCIA	Más lento en ejecución	Más rápido desde el punto de vista del hardware
ESPACIO OCUPADO	Ocupa menos espacio, cada sentencia se deja en la forma original	Una sentencia puede ocupar cientos de sentencias de máquina
DETECCIÓN DE ERRORES	Las sentencias del código fuente pueden ser relacionadas directamente con la que se está ejecutando	Cualquier referencia al código fuente se pierde en el código objeto

- **Combinación de ambas técnicas**
  1) Compiladores e intérpretes se diferencian en la forma que reportan errores de ejecución. Algunos ambientes usan las dos versiones: en la etapa de desarrollo utilizan INTÉRPRETE (diagnóstico de errores). Post validación se COMPILA para generar código eficiente.
  2) Traducción a código intermedio que luego se interpreta. Genera código portable. Ejemplo: Java.
- **Compiladores:** pueden ejecutarse en un solo paso o en dos. La ejecución de los programas es más rápida. Ejemplos: C, Ada, Pascal. Poseen varias etapas:
  - Análisis
    - Análisis léxico (Scanner): Hace análisis a nivel de palabra. Filtra comentarios, espacios en blanco, tabulaciones. Genera errores si la entrada no coincide con categorías léxicas. Analiza el tipo de cada token. Es el que lleva más tiempo.
    - Análisis sintáctico (Parser): Hace análisis a nivel de sentencia. Se alterna con el análisis semántico. Se identifican estructuras: sentencias, declaraciones, expresiones, etc. Aplica gramática para construir árbol sintáctico del programa.
    - Análisis semántico (Semántica estática): Fase medular y la más importante. Estructuras sintácticas son procesadas y la estructura del código ejecutable toma forma. Comprobación de tipos. Comprobación de nombre. *Nexo entre el análisis y la síntesis*.
  - Generación de código intermedio
  - Síntesis
    - Optimización y generación del código: Se construye el programa ejecutable. Se genera código necesario y se optimiza (optativo).



Teoría 4: Variable

Los programas tienen *entidades* : variable, rutina, sentencia.
Las entidades tienen *atributos*, que tienen que asociarse antes de poder usar la entidad. Esta asociación entre entidad y atributo se denomina **ligadura**.

Momentos y estabilidad de ligadura (binding time)

- Estática: se establece antes de la ejecución y no se puede cambiar.
- Dinámica: se establece en ejecución y puede cambiar.

Características:

- Nombre (identificador) conjunto de caracteres para referenciar a la variable. En cada lenguaje puede cambiar su longitud máxima, los caracteres aceptados y su sensibilidad a mayúsculas y minúsculas.
- Alcance: rango de instrucciones en el que se conoce y se manipula la variable a través de operaciones.
    - Estático o léxico: se define el alcance en términos de la estructura léxica del programa. Se puede ligar una variable estáticamente en la declaración, examinando el texto del programa. (en C, en ADA, en Pascal y en Python)
    - Dinámico: se define el alcance en ejecución. La declaración extiende su efecto sobre las instrucciones posteriores hasta que una nueva declaración con el mismo nombre sea encontrada
    - Local: referencias creadas dentro de un programa o subprograma.
    - No local: referencias usadas dentro de un subprograma pero creadas fuera de él.
    - Global: referencias creadas en el programa principal.
    - Espacio de nombres: zona de memoria separada donde se pueden declarar y definir objetos y funciones.
- Tipo: conjunto de valores y sus operaciones.
    - Momento de ligadura:
        - Estático: se liga en compilación y no puede cambiar.
            - Explícito: se establece mediante una declaración.
            - Implícito: la ligadura se deduce por reglas.
            - Inferido: se deduce de los tipos de sus componentes.
        - Dinámico: se liga en ejecución y puede cambiar.
    - Predefinidos: tipos base descriptos en la definición.
    - Definidos por el usuario: declarar y definir nuevos tipos a partir de los predefinidos.
    - Tipos de datos abstractos (TAD): se debe especificar su representación y sus operaciones.
- L-valor: lugar o celda de memoria asociado a la variable (ttl).
    - Momento de alocación:
        - Estática: sensible a la historia

        - Dinámica:
            - automática: cuando se declara.
            - explícita: a través de un constructor.
        - Persistente: su tiempo de vida no depende de la ejecución.
- R-valor: valor almacenado en la celda de memoria.
    - Momentos:
        - Dinámico: por naturaleza.
        - Constante: se define un valor que no puede cambiar.
    - Inicialización: por defecto (en 0) o en la declaración.

- Variable referencia: que el r-valor de una variable sea una referencia al l-valor de otra.
- Alias: dos nombres que denotan la misma entidad. Comparten el mismo objeto en el mismo ambiente de referencia, sus caminos de acceso conducen al mismo objeto.
- Sobrecarga: un nombre apunta a diferentes entidades.

Teoría 5: Unidades de programa

Un programa está compuesto por unidades, en general llamadas rutinas.

- **Nombre:** identificador para invocar a la rutina. Se introduce en su declaración.
- **Alcance:** rango de instrucciones donde se conoce. La llamada a la rutina (activación) puede estar sólo dentro del alcance.
- **Tipo:** en el encabezado se define el tipo de parámetros y el tipo de valor de retorno.
- **L-valor:** lugar de memoria donde se almacena el cuerpo de la rutina.
- **R-valor:** la llamada a una rutina causa la ejecución de su código, esto forma su r-valor. El uso de punteros a rutinas permite una política dinámica de invocación de rutinas.

Si el lenguaje permite distinguir entre declaración y definición de una rutina, permite manejar esquemas de rutinas mutuamente recursivas.

Comunicación entre rutinas: parámetros

- **Parámetros formales:** aparecen en la definición de la rutina.
- **Parámetros reales:** aparecen en la invocación de la rutina.

Ligadura entre parámetros:

- **Método posicional:** se ligan uno a uno de acuerdo a su posición.
- **Método por nombre:** deben conocerse los nombres de los formales.

Instancia de la unidad: representación de la rutina en ejecución.
Rutina como proceso de cómputo. Cuando se invoca, se ejecuta una instancia del proceso con parámetros definidos.

- Segmento de código: instrucciones almacenadas en memoria.
- Registro de activación: datos locales de la unidad almacenada.

Elementos de ejecución

- **Punto de retorno:** debe ser guardada en el registro de activación.
- **Ambiente local:** variables locales
- **Ambiente no local:** variables no locales

Estructura de ejecución

- **Estática - espacio fijo:** el espacio necesario para la ejecución se deduce del código, se conoce antes de la ejecución.
- **Basada en pila - espacio predecible:** el espacio a utilizarse es predecible y sigue una disciplina last-in-first-out. Se usa una pila.
- **Dinámica - espacio impredecible:** los datos se alocan dinámicamente en la zona de memoria heap, cuando se los necesita durante la ejecución.

Procesador abstracto: SIMPLESEM

Se traducen las reglas de cada constructor del lenguaje a una secuencia de instrucciones equivalentes.

- **Memoria de código:** C(y) valor almacenado en una celda de la memoria de código.
- **Memoria de datos:** D(y) valor almacenado en una celda de memoria de datos.
- **IP:** puntero a la instrucción que se está ejecutando.

Instrucciones:

- **set**: setea los valores en la memoria de datos
  set target,source
- **jump**: bifurcación incondicional
- **jumpt**: bifurcación condicional, evalúa una expresión como verdadera.

C1: lenguaje simple
- Sentencias y tipos simples
- Sin funciones
- Datos estáticos

C2: C1 + rutinas internas
- rutinas internas disjuntas

- no son recursivas

C2': rutinas compiladas separadas

Teoría 6: Formas de comunicación entre las rutinas

Rutinas: sentencias que representan acción abstracta. Representan una unidad de programa. Sinónimo de subprograma. Permite definir una operación creada por el usuario. Su invocación representa el uso de dicha abstracción. Cuando se usa el programa se ignora el COMO y se concentra en el QUÉ puede hacer -> implementación oculta.

- **Formas de subprogramas**
  - Procedimientos: definen *nuevas sentencias* creadas por el usuario. Resultados en variables no locales o parámetros.
  - Funciones: definen un *nuevo operador*. Sólo producen un valor. Este valor reemplaza a la invocación dentro de la expresión.
- **Definición de subprograma:** especificación o encabezado, implementación.
- **Uso de subprograma:** se lo invoca por su nombre junto con la lista de argumentos y generando una instancia de la unidad.

Parámetros: forma de compartir datos entre diferentes unidades (puede ser también a través de variables globales -> acceso a ambiente no local).

- **Pasaje de parámetros:** el más flexible y permite transferencia de diferentes datos en cada llamada. Ventajas en legibilidad y modificabilidad. Permiten compartir los datos en forma abstracta.
- **Lista de parámetros:** conjunto de datos que se van a compartir.
  - Parámetros reales: se codifican en la invocación del subprograma. Puede ser local a la unidad llamadora, o parámetro formal de ella, o un dato no local, pero visible en dicha unidad.
  - Parámetros formales: se declaran en la especificación del subprograma. Son similares a las variables locales. Contiene nombres y tipos de los datos compartidos.
- **Clases de parámetros:** Los parámetros formales se clasifican desde el punto de vista semántico en:
  - Modo IN: Parámetro formal recibe el dato desde el parámetro real.
    - Por valor: El valor del parámetro real se usa para inicializar el parámetro que corresponda al invocar la unidad. Se transfiere el dato real y actúa como una variable local de la unidad llamada. El parámetro real no se modifica.
    - Por valor constante: La implementación debe verificar que el parámetro real no sea modificado. El parámetro real no se modifica. Ejemplo: Parámetros IN de ADA.

    - Modo OUT: Parámetro formal envía el dato al parámetro real.
        - Por resultado: El valor del parámetro formal se copia al parámetro real al terminar de ejecutarse la unidad llamada. El parámetro formal es una variable loca, sin valor inicial.
        - Por resultado de funciones: El resultado de una función puede devolverse con el return en el nombre de la función que se considera como variable local. El resultado reemplaza la invocación en la expresión que contiene el llamado.
    - Modo IN/OUT: Parámetro formal recibe el dato del parámetro real y el parámetro formal envía el dato al parámetro real.
        - Por valor-resultado: Copia a la entrada y a la salida de la activación de la unidad llamadora. El parámetro formal es una variable local que recibe una copia a la entrada del contenido del parámetro real y a la salida el parámetro real recibe una copia de lo que tiene el parámetro formal.
        - Por referencia: Se transfiere la dirección del parámetro real al parámetro formal. Cualquier cambio que se realice en el parámetro formal dentro del subprograma quedará registrado en el parámetro real. Se pueden modificar los valores inadvertidamente.
        - Por nombre: El parámetro formal es sustituido textualmente por el parámetro real. Se establece la ligadura en el momento de la invocación, pero la ligadura de valor se difiere hasta el momento en que lo utiliza. Si el dato es:
            - Único valor -> Igual que pasaje por referencia
            - Constante -> Igual que por valor
            - Elemento de un arreglo -> Puede cambiar el suscripto entre distintas referencias
            - Expresión -> Se evalúa cada vez
- **Pasaje de parámetros en lenguajes**
    - Lenguaje C
        - Por valor (si se necesita referencia -> punteros)
    - Modula II y Pascal
        - Por valor (por defecto)
        - Por referencia (opcional: VAR)
    - C++
        - Igual que C más pasaje por referencia
    - Java
        - Único mecanismo contemplado -> Copia de valor. Pero en realidad constituye un paso por referencia de la variable por las variables de tipo NO primitivos.
    - Python
        - Pasaje por valor, pero si se pasa un objeto “mutable” (listas) se trabaja sobre él

    - ADA
        - Por copia IN (por defecto)
        - Por resultado OUT
        - IN OUT: para los primitivos -> indica por valor-resultado. Para los no primitivos, datos compuestos -> se hace por referencia.
        - En las funciones -> solo paso por copia de valor.

- **Subprogramas como parámetro:** A veces es conveniente manejar como parámetros los nombres de los subprogramas. Se comportan muy distinto en lenguajes con reglas de alcance estático que dinámico.
    - Ligadura shallow/superficial: El ambiente de referencia es el del subprograma que TIENE el parámetro formal subprograma. No es apropiada para lenguajes con estructuras de bloques.
    - Ligadura deep/profunda: El ambiente de referencia es el del subprograma donde ESTÁ declarado el subprograma usado como parámetro real. Los lenguajes de alcance estático utilizan esta ligadura. Se necesita que en el momento de la invocación a la unidad con parámetro subprograma se indique cual debe ser su ambiente de referencia (puntero al lugar de la cadena estática).
    - El ambiente del subprograma donde se encuentra el llamado a la unidad que tiene un parámetro subprograma. Poco común.
- **Unidad genérica**: unidad que puede instanciarse con parámetros formales de distinto tipo. Por ejemplo: unidad genérica que ordena un arreglo puede instanciarse para enteros, flotantes, etc.

Teoría 7: Tipos de datos

Introducción a sistema de tipos: abstracción de datos en los lenguajes de programación. Trata con las componentes de un programa que son sujeto de computación. Está basada en las propiedades de los objetos de datos y las operaciones de dichos objetos.

Sistema de tipos: Conjunto de reglas que estructuran y organizan una colección de tipos. Objetivo -> Lograr que los programas sean tan seguros como sea posible.

- **Tipados**
    - Tipado fuerte: especifica restricciones sobre como las operaciones que involucran valores de diferentes tipos pueden operarse.
    - Tipado débil: no especifica restricciones.
- **Especificación**
    - Tipo y tiempo de chequeo
        - Tipos de ligadura

        - Tipado estático: ligaduras en compilación. “El tipado es estático si cada ENTIDAD queda ligada a su tipo durante la compilación, sin necesidad de ejecutar el programa”. “El tipado es estático si cada VARIABLE queda ligada a su tipo durante la compilación, sin necesidad de ejecutar el programa”.
        - Tipado dinámico: ligaduras en ejecución, provoca más comprobaciones en tiempo de ejecución
    - Tipado seguro: no es estático ni inseguro
  - Reglas de equivalencia y conversión
  - Reglas de interferencia de tipo
  - Nivel de polimorfismo del lenguaje
- **Lenguajes fuertemente tipados:** si el sistema de tipos impone restricciones que aseguran que no se producirán errores de tipo de ejecución. Compilador -> Garantiza ausencia de errores de tipo en los programas. Todos los errores de tipo se detectan.

Tipo de dato: capturan la naturaleza de los datos del problema que serán manipulados por los programas.

- **Definición:**
  - Desde punto de vista Denotacional -> conjunto de valores sobre un dominio
  - Desde punto de vista Constructivo -> primitivo provisto por el lenguaje. Compuesto empleando constructores de tipo.
  - Desde punto de vista de Abstracción -> conjunto de operaciones con semántica bien definida y consistente.
- **Operaciones:** única forma de manipular los objetos instanciados
- **Tipo:** comportamiento abstracto de un conjunto de objetos y un conjunto de operaciones.

	Elementales	Compuestos
Predefinidos	Enteros Reales Caracteres Booleanos	String
Definidos por el usuario	Enumerados	Arreglos Registros Listas

Nueva estructura -> TIPO

Nuevo comportamiento -> TAD

Tipos predefinidos: reflejan el funcionamiento del hardware subyacente y son una abstracción de él. Números (enteros, reales), caracteres, booleanos. Ventajas:

- Invisibilidad de la representación
- Verificación estática
- Desambiguar operadores
- Control de precisión

Tipos definidos por el usuario: Separan la especificación de la implementación -> definir nuevos tipos e instanciarlos. Chequeo de consistencia. Los detalles de la implementación sólo quedan en la definición del tipo.

- Enumerativos
    - Dominio: Lista de constantes simbólicas.
    - Operaciones: Comparación, asignación, posición en la lista.
      TYPE verano IS (ENERO, FEBRERO, MARZO). X: verano.
- Subrangos
    - Dominio: Subconjunto de un tipo entero o de un enumerado.
    - Operaciones: Hereda las operaciones del tipo original.
      TYPE verano = ENERO..MARZO
- Compuestos – Constructores
    - Constructores: mecanismo que proveen los lenguajes para agrupar datos denominados compuestos.
    - Dato compuesto: nombre único. Posibilidad de manipular un conjunto completo.
    - Tipo compuesto: nuevos tipos definidos por el usuario usando los constructores.
    - Rutinas: constructores que permiten combinar instrucciones elementales para formar un nuevo operador.
    - Tipos:
        - Producto cartesiano: Registros
        - Correspondencia finita: Arreglos.
        - Unión/Unión discriminada: define un tipo como la disyunción de los tipos dados. Manipula diferentes tipos en distinto momento de la ejecución. Chequeo dinámico. La unión discriminada agrega un discriminante para indicar la opción elegida.
        - Recursión: estructura que puede contener componentes del tipo T. Define datos agrupados cuyo tamaño puede crecer arbitrariamente, y cuya estructura puede ser arbitrariamente compleja.
        - Se implementan a través de PUNTEROS
            - PUNTEROS: referencia a un objeto. Variable puntero es variable cuyo r-valor es una referencia a un objeto. *Inseguridad de los punteros:*
                - Violación de tipos: variable puntero <> tipo de

dato en memoria

- Referencias sueltas: puntero que contiene una dirección de una variable dinámica que fue desalocada.
- Liberación de memoria – Objetos perdidos: los objetos apuntados son alocados en la heap. La heap puede agotarse. Si los objetos en la heap dejan de ser accesibles esa memoria podría liberarse (dispose-delete | garbage collector).
- Punteros no inicializados: peligro de acceso descontrolado a posiciones de memoria.
    - Punteros y Unión discriminada en C: (Preguntar)
    - Alias: El puntero y el valor apuntado son alias.

Teoría 8: Tipos de datos recursivos TADs

Abstracción: representar algo con sus características esenciales.

TAD: Representación(datos) + Operaciones (funciones y procedimientos)

- **Encapsulamiento:** la representación y las operaciones se describen en una única unidad sintáctica.
- **Ocultamiento de la información:** la representación y la implementación del tipo permanecen ocultos.
-

Simula-67: class sin ocultamiento de información
Modula-2: módulo
ADA: paquete - package
C++: clase
Java: clase

Especificación: conjunto de axiomas que describen el comportamiento de las operaciones.

TAD nombre-tipo (valores)

Ejemplos:

- **Conjunto:** colección de elementos sin duplicidades, en cualquier orden. Sirve para representar conjuntos matemáticos.

- **Pila** en ADA:

```
package PILA is
  type PILA limited private
  MAX: constant: 100
 function EMPTY()
 procedure PUSH()
 …
   private
      type PILA is
       vector: array(1..max) of integer
       …
end PILA
package body PILA is  //implementación
    function EMPTY() …
    procedure PUSH() …
end PILA
```

- **Clase:** tipo definido por el usuario. Contiene la especificación de los datos que describen un objeto junto con la descripción de sus acciones.

```
class Punto
 { private int x;
   private int y;
   public Punto(int x, int y) //constructor
    { x= x; y=y; }
   public Punto() //constructor sin argumentos
    { x=y=0; }
  public int leerX()
  { return x; }
    … // más operaciones
  }
```

Teoría 9: Estructuras de control - Excepciones

Estructuras de control: medio por el cual los programadores pueden determinar el flujo de ejecución entre los componentes de un programa.

- **A nivel de unidad:** cuando el flujo de control se pasa entre unidades. Intervienen los pasajes de parámetros
- **A nivel de sentencia:**
  - Secuencia: Flujo de control más simple. Indica la ejecución de una sentencia a continuación de otra. Delimitador general: “;”. Lenguajes orientados a línea -> por cada linea una instrucción. Se pueden agrupar sentencias con delimitadores como “begin/end” o llaves.
  - Selección: permite que el programador pueda expresar una elección entre un cierto número posible de sentencias alternativas.

    IF (condición lógica) sentencia -> Fortran

Otros ejemplos: En C, no lleva el then. Si es más de una sentencia se encierra entre llaves. En Python, se usan : al final del if, else y elif. La indentación es obligatoria al colocas las sentencias correspondientes. Sin ambigüedad y legible. Python también permite sentencia condicional (primero la sentencia, y luego la condición).

- Selección múltiple:
  - PL/1: Incorpora sentencia de selección entre dos o más opciones

```
Select
    when(A) sentencia1;
    when(B) sentencia2
    ..........
    Otherwise sentencia n;
End;
```

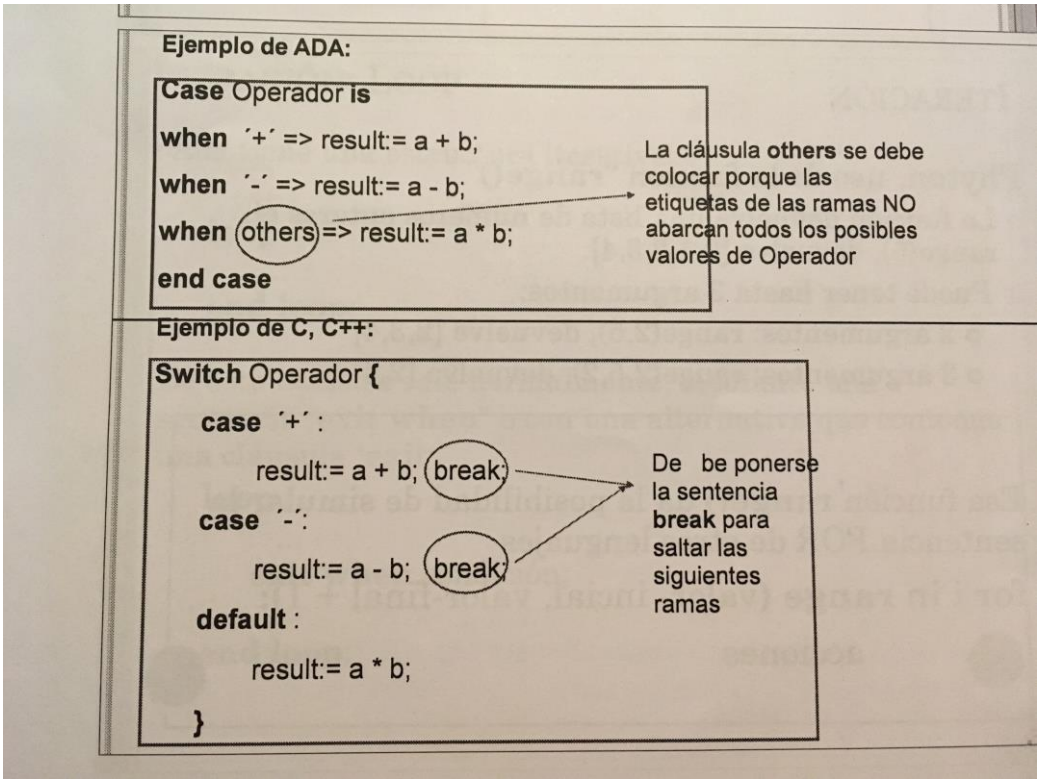
  - Algol68: La expresión debe ser de tipo entera. Tiene cláusula opcional OUT y se ejecuta cuando valor dado por la expresión no está expresado en el conjunto de valores.
  - Pascal: Es inseguro porque no establece qué sucede cuando un valor no cae dentro de las alternativas.

```
var opcion : char;
   begin
    readln(opcion);
 case opcion of
          '1' : nuevaEntrada;
          '2' : cambiarDatos;
          '3' : borrarEntrada
        else writeln('Opcion no valida!!')
   end
 end
```

  - ADA: Se debe estipular todos los valores posibles que puede tomar la expresión. Cláusula Others -> se

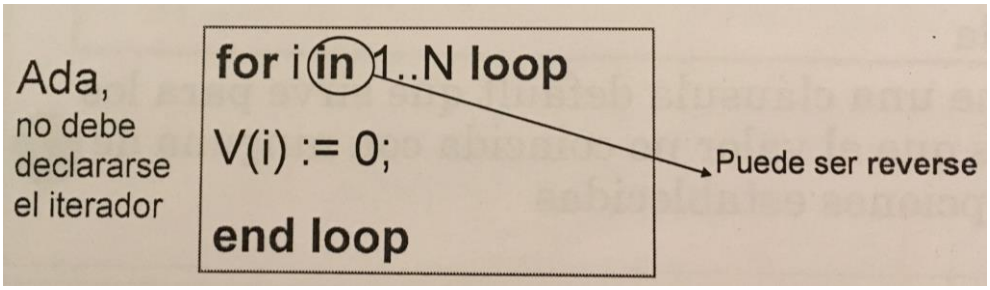
utiliza para representar aquellos valores que no se especificaron explícitamente. NO PASA LA COMPILACIÓN SI NO ESTÁ EL VALOR EXPLÍCITO NI EL OTHERS.

- C | C++: Provee el constructor switch. Break -> provoca la salida. Default -> sirve para casos en los que el valor no coincida con ninguna de las opciones.

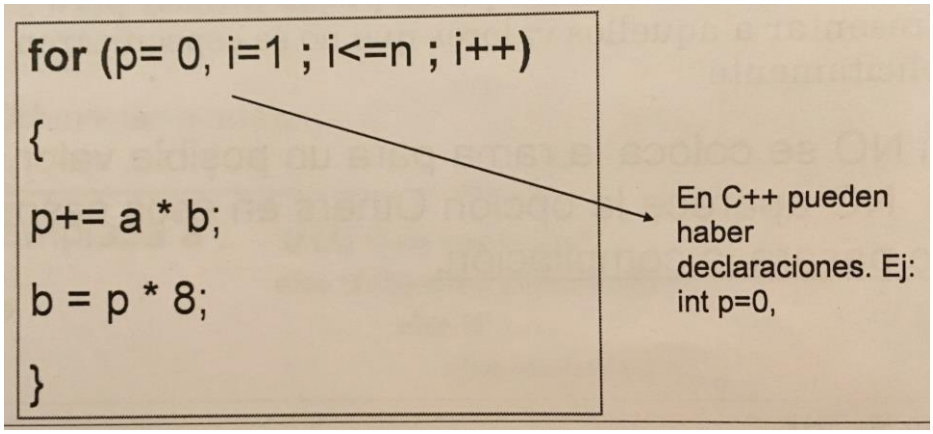


- Iteración: tipo de instrucciones que se utilizan para representar acciones que se repiten un cierto número de veces. Hay dos tipos: FOR | WHILE. Ejemplo: en Fortran original se evaluaba si la variable había llegado al límite al final del bucle (lo ejecutaba mínimo una vez).

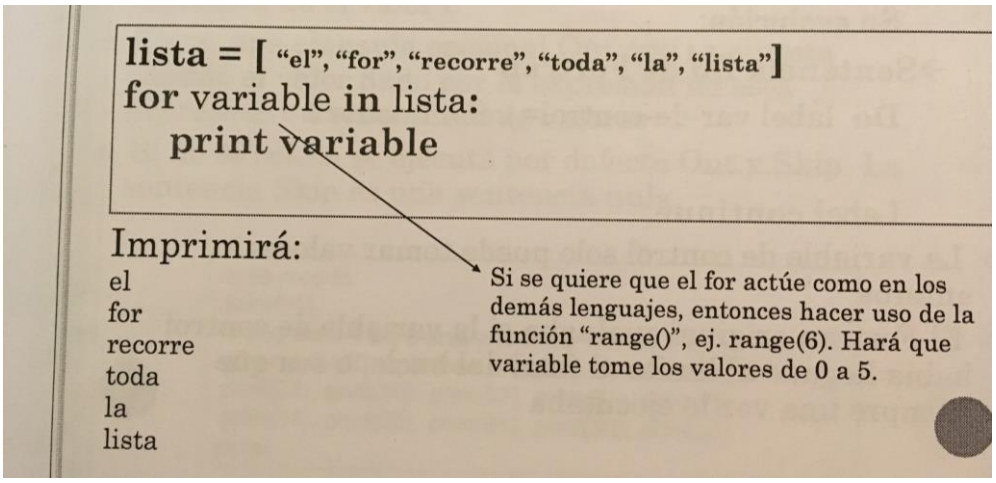
    - Sentencia FOR de Pascal, ADA, C, C++: El valor de la variable fuera del bloque se asume indefinida. La variable puede tomar cualquier valor ordinal, no solo enteros.

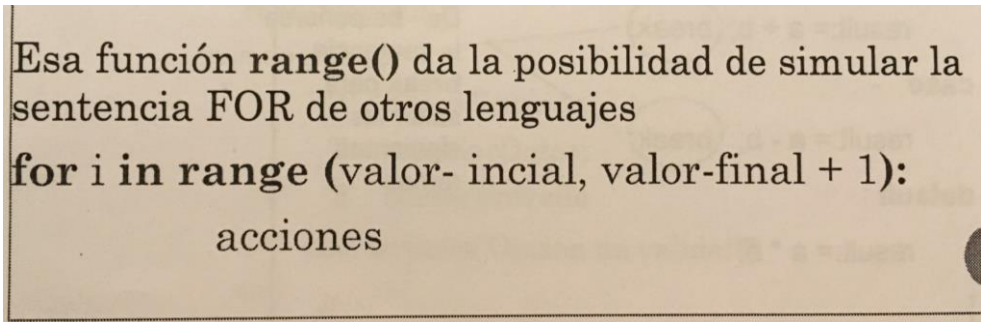


- C, C++ se compone de tres partes: una inicialización y dos expresiones.



- Python: permite iterar sobre una secuencia. Pueden ser: una lista, una tupla, etc. También utiliza la función RANGE() -> devuelve una lista de números enteros. Range(5) devuelve [0,1,2,3,4].





WHILE: Estructura que permite repetir un proceso mientras se cumpla una condición. La condición se evalúa antes de que entre al proceso.

- Pascal: while condición do sentencia;

- C | C++: while (condición) sentencia;
- ADA: while condición sentencia end loop;
- Phyton: while condicion: sentencia 1… sentencia N

UNTIL: Estructura que permite repetir un proceso mientras se cumpla una condición. La condición se evalúa al final del proceso, por lo menos una vez el proceso se realiza.

- Pascal: repeat sentencia until condición;
- C | C++: do sentencia; while (condición);
- ADA: loop … end loop; -> se sale mediante un “exit when” o con alternativa que contenga cláusula “exit”.

Excepciones: proceso anómalo. Condición inesperada o inusual durante la ejecución de un programa.
La unidad en donde se provocó está incapacitada para atenderlo de manera que termine normalmente. Puede usarse un mecanismo de gestión de excepciones de los lenguajes ó se puede simular.
Ejemplos: división por cero, abrir archivo que no existe, operación no válida para un tipo de dato, índices no existentes en un arreglo, etc.

- **Reasunción:** Se maneja la excepción y se devuelve el control al punto siguiente donde se invocó a la excepción -> continuación.
- **Terminación:** Se termina la ejecución de la unidad que alcanza la excepción y se transfiere el control al manejador.

Excepciones ya definidas por el lenguaje -> built-in

Excepciones por lenguaje

LENGUAJE	ADA
MODELO	Terminación
DECLARACIÓN	Se declaran en zona de declaración de variables
ALCANCE DE LA DECLARACIÓN	Mismo alcance que las variables
ELECCIÓN DEL MANEJADOR	Se busca manejador al final del bloque en ejecución. Si existe en ese ámbito -> le cede el control. Si no existe -> se propaga la excepción siguiendo cadena DINÁMICA (multinivel): se levanta en otro ámbito. Si se quiere continuar ejecutando instrucciones (no terminación), es necesario crear un bloque más interno.

DECLARACIÓN DEL MANEJADOR	Al final de bloque, procedimiento, paquete o tarea begin <sentencias> exception when <nomE> -> <cuerpo> end
PARÁMETROS A MANEJADORES	Sólo parámetros de tipo String
MECANISMO PARA LLAMARLAS	raise <nomE>. Si no se especifica un nombre, se levanta nuevamente la última.

LENGUAJE	CLU
MODELO	Terminación
DECLARACIÓN	Se declaran en encabezado de procedimientos. <sentencia> except when Excep1: Manejador1; when Excep2: Manejador2; when Others: ManejadorN;
ALCANCE DE LA DECLARACIÓN	Son alcanzadas por los procedimientos.
ELECCIÓN DEL MANEJADOR	Se finaliza el procedimiento donde se originó la excepción y se devuelve el control al llamante inmediato donde se busca un manejador. Si lo encuentra en ese ámbito-> Ejecuta y se continúa con la instrucción siguiente al conjunto de manejadores. Si no lo encuentra en ese ámbito -> Se propaga ESTÁTICAMENTE. Si no lo encuentra en el procedimiento que hizo la llamada -> Excepción “failure” y se devuelve el control, terminando todo el programa.
DECLARACIÓN DEL MANEJADOR	A nivel de sentencia, dentro de procedimientos: <sentencia> except when <nomE>: <cuerpo> when others: <porDefecto>

PARÁMETROS A MANEJADORES	Parámetros de cualquier tipo
MECANISMO PARA LLAMARLAS	signal <nomE>. Mecanismo para volver a levantar excepción -> resignal

LENGUAJE	PL/1
MODELO	Reasunción
DECLARACIÓN	Se pueden declarar excepciones definidas por el usuario
ALCANCE DE LA DECLARACIÓN	Se pueden deshabilitar excepciones incorporadas al lenguaje -> Se antepone un NO+nombreExcepción. El alcance de un manejador termina cuando finaliza la ejecución de la unidad donde fue declarado.
ELECCIÓN DEL MANEJADOR	Se produce una excepción -> se ejecuta el manejador y devuelve el control a la sentencia siguiente donde se levantó. Definición de manejadores sigue política de pila. Se elige el último manejador en la pila apropiado a dicha excepción -> dinámicamente
DECLARACIÓN DEL MANEJADOR	A nivel de sentencia: <sent> on condition (<NomE>) begin <cuerpo> end
PARÁMETROS A MANEJADORES	No se pueden parametrizar los manejadores.
MECANISMO PARA LLAMARLAS	signal condition (<nomE>).

LENGUAJE	C++
MODELO	Terminación
DECLARACIÓN	Opcional poner junto a “try” los nombres de las excepciones definidas por el usuario que se pueden elevar.

ALCANCE DE LA DECLARACIÓN	
ELECCIÓN DEL MANEJADOR	Mecanismo similar al de ADA. Si la rutina alcanzó otra excepción que no está contemplada en el listado -> No se propaga la excepción: se ejecuta unexpected() que generalmente aborta el programa. Si no hay listado de posibles excepciones -> Se propaga la excepción. Si hay una lista vacía en la interfaz -> Ninguna excepción es propagada.
DECLARACIÓN DEL MANEJADOR	Asociados a bloques. try <sentencias> catch <manejadores>
PARÁMETROS A MANEJADORES	Parámetros de cualquier tipo.
MECANISMO PARA LLAMARLAS	throw <nombreExcepcion>

LENGUAJE	JAVA
MODELO	Terminación
DECLARACIÓN	Se declaran nuevas excepciones, creando clases que implementen interfaz “Throwable”
ALCANCE DE LA DECLARACIÓN	
ELECCIÓN DEL MANEJADOR	Mecanismo similar al de ADA. Todas las excepciones que puedan ser alcanzadas explícitamente, deben ser listadas en la interfaz de la misma o manejador.
DECLARACIÓN DEL MANEJADOR	try <sentencias> catch <manejadores>
PARÁMETROS A MANEJADORES	Parámetros de cualquier tipo y cantidad.
MECANISMO PARA LLAMARLAS	throw <nombreExcepcion>

LENGUAJE	PHYTON
MODELO	Terminación
DECLARACIÓN	
ALCANCE DE LA DECLARACIÓN	Si una excepción no encuentra su manejador en su bloque try except -> Busca estáticamente. Analiza si está contenido dentro de otro y si ese otro tiene el manejador. Sino -> Busca dinámicamente. Si no se encuentra -> corta el proceso.
ELECCIÓN DEL MANEJADOR	Conjunto de excepciones pueden ser manejadas por el mismo manejador. Puede haber un except sin nombre pero sólo al final. Actúa como comodín. Else -> Se ejecuta SOLO SI no se levantó una excepción Finally -> Se ejecuta SIEMPRE.
DECLARACIÓN DEL MANEJADOR	try <sentencias> except <exc1> except <exc2> else (opcional): finally (opcional):
PARÁMETROS A MANEJADORES	
MECANISMO PARA LLAMARLAS	raise (nomE)

Teoría 10: Paradigmas de programación

Paradigma de programación: estilo de desarrollo de programas. Modelo para resolver problemas computacionales. Los lenguajes de programación se encuadran en uno o varios, tiene una relación directa con su sintaxis.

Principales paradigmas:

- **Imperativo:** sentencias + secuencias de comandos
- **Declarativo:** describen los resultados esperados sin listar explícitamente los pasos a llevar a cabo para alcanzarlos
- **Lógico:** Aserciones lógicas -> hechos + reglas
- **Funcional:** Los programas se componen de funciones
- **Orientado a Objetos:** Métodos + mensajes

De aparición reciente (no forman parte de los principales):

- **Dirigido por eventos:** El flujo del programa está determinado por sucesos externos (acción del usuario)
- **Orientado a aspectos:** apunta a dividir programa en módulos independientes, cada uno con comportamiento definido.

Paradigma funcional: basado en el uso de funciones. Popular en: inteligencia artificial, matemática, lógica, procesamiento paralelo.

Características de LENGUAJES funcionales:

- Define conjunto de datos
- Provee conjunto de funciones primitivas y de formas funcionales
- Requiere operador de aplicación

Características de PROGRAMAS funcionales:

- Semántica basada en valores
- Regla de mapeo basada en combinación o composición
- Funciones de primer orden
- Transparencia referencial

PROGRAMACIÓN FUNCIONAL

Función: valor más importante en la programación funcional

f: A -> B (a cada elemento de A le corresponde un único elemento de b).

Las funciones son tratadas como valores y pueden ser pasadas como parámetros, retornar resultados, etc.

Se debe distinguir el VALOR y la DEFINICIÓN de una función. Existen muchas maneras de definir una función. El tipo de una función puede estar definida explícitamente dentro del script, o puede deducirse/inferirse.

Expresiones y valores: la expresión es la noción central de la programación funcional. Característica: Una expresión es su VALOR. El valor de la expresión depende únicamente de los valores de las sub expresiones que la componen. Dichas expresiones pueden contener variables (valores desconocidos).

Variable -> variable matemática, no celda de memoria.

Expresiones -> propiedad de “transparencia referencial” -> dos expresiones que sintácticamente iguales darán el mismo valor, porque no existen efectos laterales.

Un script es una lista de definiciones y:

- Pueden someterse a evaluación
- Pueden combinarse
- Pueden modificarse

Expresiones canónicas: expresiones que pueden NO llegar a reducirse del todo, pero se les asigna un valor indefinido. Toda expresión siempre denota un valor.

Forma de evaluar las expresiones:

- **Reducción**
  - Orden aplicativo (SIEMPRE evalúa los argumentos)
  - Orden normal (no calcula más de lo necesario. No es evaluada hasta que su valor se necesite).
- **Simplificación**

Tipos (toda función tiene asociado un tipo):

- **Básicos** -> son los primitivos. Ejemplo: NUM, BOOL, CHAR.
- **Derivados** -> se construyen de otros tipos

Función identidad -> id x = x (expresión de tipo polimórfica, en dónde no es tan fácil deducir su tipo)

Currificación: mecanismo que reemplaza argumentos estructurados por más simples.

Cálculo Lambda: modelo de computación para definir funciones. Se utiliza para entender elementos de programación funcional y semántica subyacente. Expresiones pueden ser de tres clases:

- Un identificador o una constante
- Una definición de una función
- Una aplicación de una función

PROGRAMACIÓN LÓGICA

- Un programa es una serie de aserciones lógicas.
- El conocimiento se representa a través de **reglas** y **hechos**.

Elementos:

- Los objetos son representados por **términos**.
- Variables: son elementos indeterminados que pueden sustituirse. Los nombres de variables comienzan con mayúscula.

- Constantes: son elementos determinados.
- Término compuesto: consiste en un “functor” seguido de un número de argumentos.
- Estructura: término compuesto con argumentos variables.
- Listas:
    - El functor “.” construye una lista de un elemento y una lista. Ejemplo: .(alpha, []) o .[alpha | []]
- **Cláusulas de Horn:**
    - Programa: secuencia de “cláusulas”.
    - Las cláusulas pueden ser hechos o reglas.
    - Hecho: expresa relaciones entre objetos - verdades.
    - Regla:
        - tiene la forma: conclusion :- condición
        - conclusión es un predicado.
        - ‘:-’ significa si.
        - condición es una conjunción de predicados.

PROGRAMAS Y QUERIES

Programa: conjunto de cláusulas.
Query: representa lo que deseamos que sea contestado.

Un programa es un conjunto de reglas y hechos que proveen una especificación declarativa de que es lo que se conoce y la pregunta (query) es el objetivo que queremos alcanzar.

PROGRAMACIÓN ORIENTADA A OBJETOS

Programa OO: conjunto de objetos que interactúan mandandose mensajes. Los elementos que intervienen son:

- **Objetos:** entidades que poseen estado interno y comportamiento. Equivalente a dato abstracto.
- **Mensajes:** petición de un objeto a otro para que éste se comporte de una determinada manera.
- **Métodos:** programa que está asociado a un objeto determinado y cuya ejecución sólo puede desencadenarse a través de un mensaje recibido por éste o por sus descendiente.
- **Clases:** Tipo definido por el usuario que determina las estructuras de datos y operaciones asociadas con ese tipo. Objeto -> pertenece a clase -> hereda su funcionalidad. Información contenida en el objeto puede ser accedida por la ejecución de los métodos correspondientes.
    - Instancia de clase: Construcción de objeto -> Instancia de la clase: objeto individualizado por los valores que tomen sus atributos.

GENERALIZACIÓN/ESPECIFICACIÓN => HERENCIA (agrupar clases en jerarquías de clases definiendo sub y super clases)

POLIMORFISMO: capacidad que tienen los objetos de distintas clases de responder a mensajes con el mismo nombre.

BINDING DINÁMICO: vinculación en el proceso de ejecución de los objetos con los mensajes.

Ejemplo: C++

- Lenguaje extendido de C incorporando características OO
- Lenguaje compilativo

Teoría 11: Lenguajes de scripting

Lenguajes de scripting: diseñados para "pegar" programas existentes a fin de construir un sistema más grande. Se utilizan como lenguajes de extensión, permiten al usuario adaptar o extender las funcionalidades. Son interpretados.

Poseen dos tipos de ancestros:

- Intérpretes de líneas de comando (shells)
- Herramientas para procesamiento de texto y generación de reportes

Asumen existencia de componentes útiles en otros lenguajes. La intención no es escribir aplicaciones desde el comienzo, sino por combinación de componentes.

Las declaraciones son escasas o nulas. Muchos de estos lenguajes incorporan tipificación dinámica.

PHP, Python, Ruby, Scheme -> Tipo de variable es chequeado inmediatamente antes de su uso.

Rexx, Perl, Tcl -> Tipo de variable es interpretado de manera diferente según el contexto.

- Útiles para:
  - Procesamiento de texto | Reportes
  - Manejar E/S de programas externos
  - Mayoría de comandos basados en Expresiones Regulares Extendidas
- Poseen tipos de datos de alto nivel: conjuntos, diccionarios, listas, tuplas, etc.
- Dominios de aplicación: Lenguaje de comandos, matemática y estadística, extensión de lenguajes, etc.