Conceptos y Paradigmas de Lenguajes de Programación

INTRODUCCION

Al *introducir, analizar y evaluar* los conceptos más importantes de los lenguajes de programación, conseguiremos:

- Adquirir habilidad de *apreciar y evaluar* lenguajes, identificando los *conceptos* más importantes de cada uno de ellos y sus *límites y posibilidades.*
- Habilidad para *elegir,* para *diseñar, implementar o utilizar* un lenguaje.
- Enfatizar la *abstracción* como la mejor forma de manejar la complejidad de objetos y fenómenos.

¿Para qué sirve los conceptos de lenguajes?

- Aumentar la capacidad para producir software.
- Mejorar el uso del lenguaje.
- Elegir mejor un lenguaje.
- Facilitar el aprendizaje de nuevos lenguajes.
- Facilitar el diseño e implementación de lenguajes.

Objetivos de diseño:

- Simplicidad y legibilidad:
    - Los lenguajes de programación deberían:
        - Poder producir programas fáciles de escribir y de leer.
        - Resultar fáciles a la hora de aprenderlo o enseñarlo.
    - Ejemplos de cuestiones que atentan contra esto:
        - Muchas componentes elementales.
        - Conocer subconjuntos de componentes.
        - El mismo concepto semántico – distinta sintaxis.
        - Distintos conceptos semánticos – la misma notación sintáctica.
        - Abuso de operadores sobrecargados.
- Claridad en los bindings:
    - Los elementos de los lenguajes de programación pueden ligarse a sus atributos o propiedades en diferentes momentos:
        - Definición del lenguaje.
        - Implementación del lenguaje.
        - En escritura del programa.
        - Compilación
        - Cargado del programa
        - En ejecución.
    - La ligadura en cualquier caso debe ser clara.
- Confiabilidad:
    - La confiabilidad está relacionada con la seguridad
        - Cheque de tipos:
            - Cuanto antes se encuentren errores menos costoso resulta realizar los arreglos que se requieran.
        - Manejo de excepciones:
            - La habilidad para interceptar errores en tiempo de ejecución, tomar medidas correctivas y continuar.
- Soporte:
    - Debería ser accesible para cualquiera que quiera usarlo o instalarlo.
        - Lo ideal sería que su compilador o intérprete sea de dominio público.
    - Debería poder implementando en diferentes plataformas.
    - Deberían existir diferentes medios para poder familiarizarse con el lenguaje: tutoriales, cursos textos, etc.
- Abstracción:
    - Capacidad de definir y usar estructuras u operaciones complicadas de manera que sea posible ignorar muchos de los detalles.
        - Abstracción de procesos y de datos.
- Ortogonalidad:

    - Significa que un conjunto pequeño de constructores primitivos, puede ser combinado en número relativamente pequeño a la hora de construir estructuras de control y datos. Cada combinación es legal y con sentido.
        - El usuario comprende mejor si tiene un pequeño número de primitivas y un conjunto consistente de reglas de combinación.
- Eficiencia:
    - Tiempo y espacio
    - Esfuerzo humano
    - Optimizable

SINTAXIS Y SEMANTICA

Un lenguaje de programación es una notación formal para describir algoritmos a ser ejecutados en una computadora.

Definiciones:

- Sintaxis: conjunto de reglas que definen como componer letras, dígitos y otros caracteres para formar los programas.
- Semántica: conjunto de reglas para dar significado a los programas sintácticamente válidos.

La definición de la sintaxis y la semántica de un lenguaje de programación proporcionan mecanismos para que una persona o una computadora puede decir si el programa es válido y si lo es, que significa.

SINTAXIS

- La sintaxis debe ayudar al programador a escribir programas correctos sintácticamente.
- La sintaxis establece reglas que sirven para que el programador se comunique con el procesador.
- La sintaxis debe contemplar soluciones a características tales como:
    - Legibilidad
    - Verificabilidad
    - Traducción
    - Falta de ambigüedad.

La sintaxis estable reglas que definen como deben combinarse las componentes básicas, llamadas “Word”, para formar sentencias y programas.

- Elementos:
    - Alfabeto o conjunto de caracteres: Tener en cuenta con qué conjunto de caracteres se trabaja sobre todo por el orden a la hora de comparaciones. La secuencia de bits que compone cada carácter la determina la implementación.
    - Identificadores: elección más ampliamente utilizada: cadena de letras y dígitos, que deben comenzar con una letra. Si se restringe la longitud se pierde legibilidad.
    - Operadores: con los operadores de suma, resta, etc. la mayoría de los lenguajes utilizan +, -. En los otros operadores no hay tanta uniformidad.
    - Comentarios: hacen los programas más legibles.
    - Palabra clave y palabra reservada: (array, do, else, if)
        - Palabra clave o keywords, son palabras claves que tienen un significado dentro de un contexto.
        - Palabra reservada, son palabras claves que además no puede ser usadas por el programador como identificador de otra entidad.
        - Ventajas de su uso:
            - Permiten al compilador y al programador expresarse claramente.
            - Hacen los programas más legibles y permiten una rápida traducción.
        - Soluciones para evitar confusión entre palabras claves e identificadores:
            - Usar palabras reservadas
            - Identificarlas de alguna manera (Ej. Algol) usa ‘PROGRAM ‘END
            - Libre uso y determinar de acuerdo al contexto.
- Estructura sintáctica:

    - Vocabulario o words: conjunto de caracteres y palabras necesarias para construir expresiones, sentencias y programas. Ej: identificadores, operadores, palabras claves, etc. *Las words no son elementales se construyen a partir del alfabeto.*
    - Expresiones: son funciones que a partir de un conjunto de datos devuelven un resultado. Son bloques sintácticos básicos a partir de los cuales se construyen las sentencias y programas.
    - Sentencias: componente sintáctico más importante. Tiene un fuerte impacto en la facilidad de escritura y legibilidad. Hay sentencias simples, estructuradas y anidadas.
- Reglas léxicas y sintácticas:
    - Reglas léxicas: conjunto de reglas para formar las "Word", a partir de los caracteres del alfabeto.
    - Reglas sintácticas: conjunto de reglas que definen como formar las "expresiones" y "sentencias".

  La diferencia entre léxico y sintáctico es arbitrario, dan la apariencia externa del lenguaje.
- Tipos de sintaxis:
    - Abstracta: se refiere básicamente a la estructura.
    - Concreta: se refiere básicamente a la parte léxica.
    - Pragmática: se refiere básicamente al uso práctico.
- ¿Cómo definir la sintaxis?:
    - Se necesita una descripción finita para definir un conjunto infinito (conjunto de todos los programas bien escritos).
    - Formas para definir la sintaxis:
        - Lenguaje natural. Ej: Fortran
        - Utilizando la gramática libre de contexto, definida por Backus y Naun: BNF. Ej: Algol
        - Diagramas sintácticos son equivalentes a BNF pero mucho más intuitivos.
- **BNF (Backus Naun Form)**
    - Es una notación formal para describir la sintaxis
    - Es un metalenguaje
    - Utiliza metasimbolos: < > ::= |
    - Define las reglas por medo de "producciones":
        - Ejemplo: < digito > ::= 0|1|2|3|4|5|6|7|8|9 (digito es "no terminal", ::= "metasimbolo" y los números son "terminales"
- Gramática
    - Conjunto de reglas finita que define un conjunto infinito de posibles sentencias validas en el lenguaje.
    - Una gramática está formada por 4-tupla
    - G = (N, T, S, P)
        - N: conjunto de símbolos no terminales
        - T: conjunto de símbolos terminales
        - S: símbolo distinguido de la gramática que pertenece a N
        - P: conjunto de producciones
- Arboles sintácticos
    - "juan un canta manta"
        - Es una oración sintácticamente incorrecta
        - No todas las oraciones que se pueden armar con los terminales son validas
    - Se necesita de un **Método de análisis (reconocimiento)** que permita determinar si un string dado es valido o no en el lenguaje: **Parsing.**
    - El **parse,** para cada sentencia construye un "**árbol sintáctico o árbol de derivación"**
    - Dos maneras de construirlo:
        - Metodo botton-up:
            - De izquierda a derecha
            - De derecha a izquierda
        - Metodo top-down:
            - De izquierda a derecha
            - De derecha a izquierda
- Producciones recursivas:

- Son las que hacen que el conjunto de sentencias descripto sea infinito
- Ejemplo de producciones recursivas:
  - <natural> :: = <digito> | <digito><digito> | ..... | <digito>.....<digito>
- Si lo planteamos recursivamente:
  - GN = (N, T, S, P)
    - N = { <natural>, <digito> } T = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
    - S = <natural>
    - P = {<natural> ::= <digito> | <digito> <natural>,
    - <digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}
- Cualquier gramática que tiene una producción recursiva describe un **lenguaje infinito**.
- Regla recursiva por la izquierda
  - La asociatividad es por la izquierda
  - El símbolo no terminal de la parte izquierda de una regla de producción aparece al comienzo de la parte derecha
- Regla recursiva por la derecha
  - La asociatividad es por la derecha
  - El símbolo no terminal de la parte izquierda de una regla de producción aparece al final de la parte derecha

- Gramáticas ambiguas:
  - Una gramática es ambigua si una sentencia puede derivarse de más de una forma.
- Gramáticas libres de contexto y sensibles al contexto: int e ; a := b + c
  - Según nuestra gramática son sentencias sintácticamente validas, aunque puede suceder que a veces no lo sea semánticamente.
    - El identificador está definido dos veces
    - No son del mismo tipo
  - Una gramática libre de contexto es aquella en la que no realiza un análisis del contexto.
  - Una gramática sensible al contexto analiza este tipo de cosas (Algol 68).
  - Otras formas de describir la sintaxis libres de contexto:
    - EBNF: esta gramatica es la **BNF extendida.**
    - Los metasimbolos que incorpora son:
      - [ ] elemento optativo puede o no estar
      - (|) selección de una alternativa
      - { } repetición
      - * cero o mas veces ; + una o mas veces
    - Ejemplo con EBNF:
      - Definición números enteros en BNF y en EBNF
      - BNF
        - <enterosig> ::= + <entero> | - <entero> | <entero>
          <entero> ::= <digito>| <entero><digito>
      - EBNF
        - <enterosig>::= [(+|-)] <digito>{<digito>}*
      - Eliminó la recursión y es mas fácil de entender
- Diagramas sintácticos (CONWAY):
  - Es un grafo sintáctico o carta sintáctica
  - Cada diagrama tiene una entrada y una salida, y el camino determina el análisis.
  - Cada diagrama representa una regla o producción
  - Para que una sentencia sea válida, debe haber un camino desde la entrada hasta la salida que la describa
  - Se visualiza y entiende mejor que BNF y EBNF.

SEMANTICA

La semántica describe el significado de los símbolos, palabras y frases de un lenguaje ya sea lenguaje natural o lenguaje informático. Ej: int vector [10]; if (a<b) max = a; else max = b;

- Tipos de semántica:
  - Semántica **Estática**:
    - No está relacionado con el significado del programa, está relacionado con las formas válidas.
    - Se las llama así porque el análisis para el chequeo puede hacerse en compilación.
    - Para describir la sintaxis y la semántica estática formalmente sirven las denominadas gramáticas de atributos, inventadas por Knuth en 1968.
    - Generalmente las gramáticas sensibles al contexto resuelven los aspectos de la semántica estática.
  - Semántica estática – Gramática de atributos
    - A las construcciones del lenguaje se le asocia información a través de los llamados "**atributos**" asociados a los símbolos de la gramática correspondiente.
    - Los valores de los atributos se calculan mediante las llamadas "**ecuaciones o reglas semánticas**" asociadas a las producciones gramaticales.
    - La evaluación de las reglas semánticas puede:
      - Generar código.
      - Insertar información en la tabla de símbolos.
      - Realizar el chequeo semántico.
      - Dar mensajes de error, etc.
    - Los atributos están directamente relacionados con los símbolos gramaticales (terminales y no terminales). La forma, general de expresar las gramáticas con atributos se escriben en forma tabular. Ej:

Regla gramatical	Reglas semánticas
Regla 1	Ecuaciones de atributo asociadas
.	.
Regla n	Ecuaciones de atributo asociadas

    - Ej. Gramática simple para una declaración de variable en el lenguaje C. Atributo at

Regla gramatical	Reglas semánticas
Decl → tipo lista-var	lista-var.at=tipo.at
Tipo → int	tipo.at = int
Tipo → float	tipo.at = float
Lista-var → id	id.at = lista-var.at Añadetipo(id.entrada, lista-var.at)
Lista-var → id, lista-var	id.at = lista-var.at Añadetipo(id.entrada, lista-var.at) Lista-var.at = lista-var.at

  - Semántica **Dinámica:**
    - Es la que describe el efecto de ejecutar las diferentes construcciones en el lenguaje de programación.
    - Su efecto se describe durante la ejecución del programa.
    - Los programas solo se pueden ejecutar si son correctos para la sintaxis y para la semántica estática.
    - ¿Cómo describe la semántica?
      - No es fácil
      - No existen herramientas estándar como en el caso de la sintaxis (diagramas sintácticos y BNF)
      - Hay diferentes soluciones formales:
        - Semántica axiomática:

        - Considera al programa como “una máquina de estados”.
        - La notación empleada es el “cálculo de predicados”.
        - Se desarrolló para probar la corrección de los programas.
        - Los constructores de un lenguaje de programación se formalizan describiendo como su ejecución provoca un cambio de estado.
        - Un estado se describe como un predicado que describe los valores de las variables en ese estado.
        - Existe un estado anterior y un estado posterior a la ejecución del constructor.
        - Cada sentencia se precede y se continúa con una expresión lógica que describe las restricciones y relaciones entre los datos.
            - Precondición
            - Poscondicion
    - Semántica denotacional:
        - Se basa en la teoría de funciones recursivas.
        - Se diferencia de la axiomática por la forma que describe los estados, la axiomática lo describe a través de los predicados, la denotacional a través de funciones.
        - Se define una correspondencia entre los constructores sintácticos y sus significados.
    - Semántica operacional:
        - El significado de un programa se describe mediante otro lenguaje de bajo nivel implementado sobre una maquina abstracta.
        - Los cambios que se producen en el estado de la maquina cuando se ejecuta una sentencia del lenguaje de programación definen su significado.
        - Es un método informal.
        - Es el más utilizado en los libros de texto.
        - PL/1 fue el primero que la utilizo.

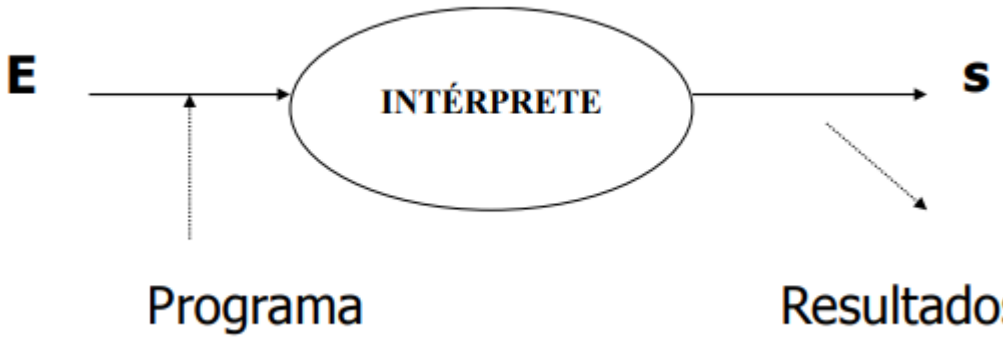
PROCESAMIENTO DE UN LENGUAJE

TRADUCCION

- Las computadoras ejecutan lenguajes de bajo nivel llamado “lenguaje de maquina”.
- Un poco de historia…
    - Programar en código de maquina
- Uso de código nemotécnico (abreviatura con el propósito de la instrucción). “lenguaje ensamblador” y “programa ensamblador”.
- Aparición de los “lenguajes de alto nivel”

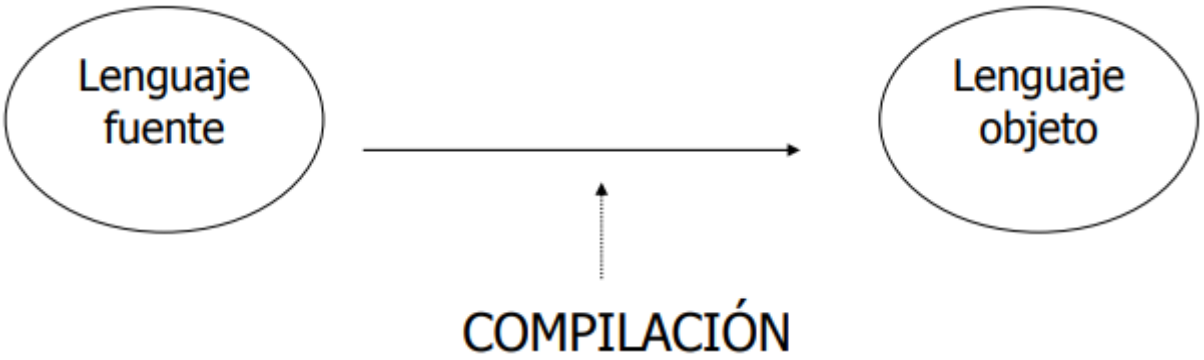
INTERPRETACION Y COMPILACION

- Interprete:
    - Lee, Analiza, Decodifica y Ejecuta una a una las sentencias de un programa escrito en un lenguaje de programación.
    - Ej: Lisp, Smalltalk, Basic, Python, etc.)
    - Por cada posible acción hay un subprograma que ejecuta esa acción.
    - La interpretación se realiza llamando a estos subprogramas en la secuencia adecuada.



- Un intérprete ejecuta repetidamente la siguiente secuencia de acciones:
  - Obtiene la próxima sentencia
  - Determina la acción a ejecutar
  - Ejecuta la acción

- Compilación:
  - Los programas escritos en un lenguaje de alto nivel se traducen a una versión en lenguaje de maquina antes de ser ejecutados.



- Traducción:
  - La compilación lleva varios pasos.
  - Ej: pasos que podría realizarse en una traducción:
    - Compilado a assembler ← Compilador
    - Ensamblado a código reubicable ← Assembler
    - Linkeditado ← Link – editor
    - Cargado en la memoria ← Loader
  - Tipos de traductores:
    - Compilador:
      - Lenguaje fuente: lenguaje de alto nivel.
      - Lenguaje objeto: cualquier lenguaje de máquina de una maquina real, o lenguaje assembler, o algún lenguaje cercano a ellos.
    - Assembler:
      - Lenguaje fuente: lenguaje assembler.
      - Lenguaje objeto: alguna variedad de lenguaje de máquina.
  - En ciertos lenguajes como C, se ejecuta antes del compilador otro traductor llamada “Macro-Procesador o Pre Procesador”
    - Macro: fragmento de texto fuente que lleva un nombre.
      - En el programa se utiliza el nombre de la macro
      - El nombre de la macro se reemplaza por su código cuando se procesen las macros.
    - Ejemplo lenguaje C: contiene directivas que deben resolverse antes de pasar a la compilación.
      - #include: inclusión de archivos de texto, Ej: #include <stdio.h>
      - #define: reemplaza símbolos por texto, Ej: #define PI 3.1416
      - Macros: funciones en-linea, Ej: #define max (x,y) ((x)>(y)?(x) : (y))

- #ifdef: compilación condicional
- El preprocesador de C frente a una macro:
  - Si se tiene la definición siguiente

    #define max(x,y) x>y?x:y
  - Y en el código aparece:

    …

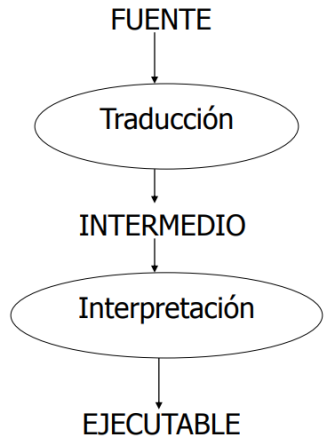
    r = max(s,5);

    ….
  - El preprocesador haría:

    ….

    r = s>5?s:5;

- Comparación entre Traductor e Interprete
  - Forma en como ejecuta:
    - Interprete: ejecuta el programa de entrada directamente.
    - Compilador: produce un programa equivalente en lenguaje objeto.
  - Forma en que orden ejecuta:
    - Interprete: sigue el orden lógico de ejecución.
    - Compilador: sigue el orden físico de las sentencias.
  - Tiempo de ejecución:
    - Interprete:
      - Por cada sentencia se realiza el proceso de decodificación para determinar las operaciones a ejecutar y sus operandos.
      - Si la sentencia está en un proceso iterativo, se realizara la tarea tantas veces como sea requerido.
      - La velocidad de proceso se puede ver afectada.
    - Compilador: no repetir lazos, se decodifica una sola vez.
  - Eficiencia:
    - Interprete: más lento en ejecución.
    - Compilador: Más rápido desde el punto de vista del hard.
  - Espacio ocupado:
    - Interprete: ocupa menos espacio, cada sentencia se deja en la forma original.
    - Compilador: una sentencia puede ocupar cientos de sentencias de máquina.
  - Detección de errores:
    - Interprete: las sentencias del código fuente pueden ser relacionadas directamente con la que se está ejecutando.
    - Compilador: cualquier referencia al código fuente se pierde en el código objeto.
- Combinación de ambas técnicas:
  - Los compiladores y los intérpretes se diferencian en la forma que ellos reportan los errores de ejecución.
  - Algunos ambientes de programación contienen las dos versiones interpretación y compilación.
    - Utilizan el intérprete en la etapa de desarrollo, facilitando el diagnostico de errores.
    - Luego que el programa ha sido validad se compila para generar código más eficiente.
  - Otra forma de combinarlos:
    - Traducción a un código intermedio que luego se interpretara.
      - Sirve para generar código portable, es decir, código fácil de transferir a diferentes maquinas.
      - Ejemplos: Java, genera un código intermedio llamado “bytecodes”, que luego es interpretado por la maquina cliente.



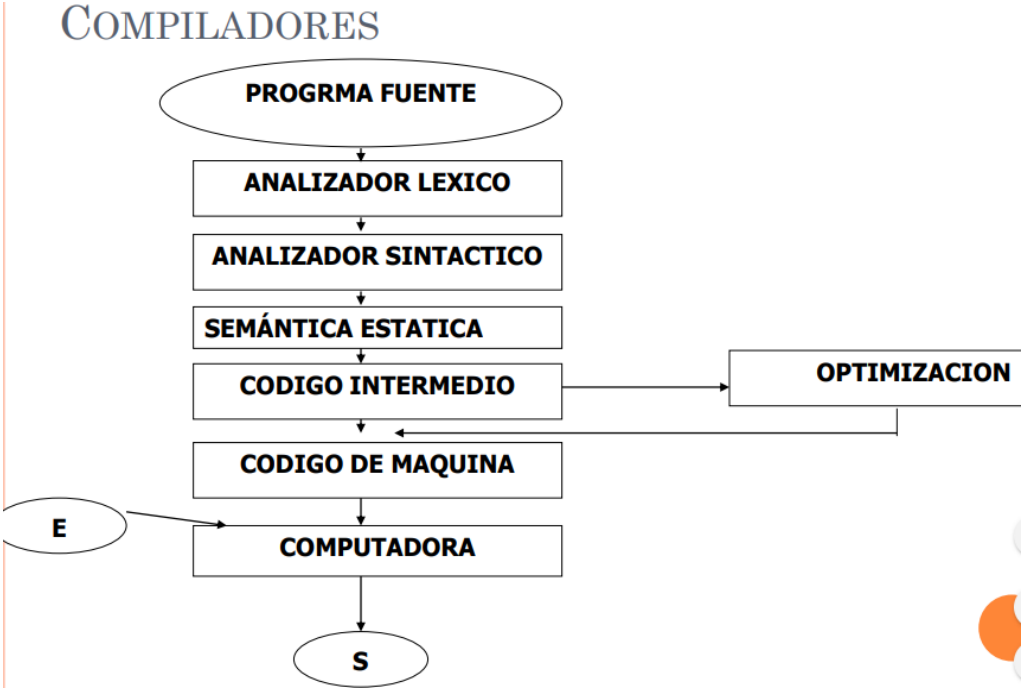
- Compiladores
  - Al compilar los programas la ejecución de los mismos es más rápida. Ej. de programas que se compilan: C, Ada, Pascal, etc.
  - Los compiladores pueden ejecutarse en un solo paso o en dos pasos.
  - En ambos casos cumplen con varias etapas, las principales son:
    - Análisis
      - Análisis léxico (Scanner):
        - Es el que lleva más tiempo.
        - Hace el análisis a nivel de palabra.
        - Divide el programa en sus elementos constitutivos: identificadores, delimitadores, símbolos especiales, números, palabras clave, delimitadores, comentarios, etc.
        - Analiza el tipo de cada token.
        - Filtra comentarios y separadores como: espacio en blanco, tabulaciones, etc.
        - Convierte errores si la entrada no coincide con ninguna categoría léxica.
        - Convierte a representación interna los números en punto fijo o punto flotante.
        - Pone los identificadores en la tabla de símbolos.
        - Reemplaza cada símbolo por su entrada en la tabla.
        - El resultado de este paso será el descubrimiento de los ítems léxicos o tokens.
      - Análisis sintáctico (Parser):
        - El análisis se realiza a nivel de sentencia.
        - Se identifican las estructuras; sentencias, declaraciones, expresiones, etc. ayudándose con los tokens.
        - El analizador sintáctico se alterna con el análisis semántico. Usualmente se utilizan técnicas basadas en gramáticas formales.
        - Aplica una gramática para construir el árbol sintáctico del programa.
      - Análisis semántico (semántica estática):
        - Es la fase medular
        - Es la más importante
        - Las estructuras sintácticas reconocidas por el analizador sintáctico son procesadas y la estructura del código ejecutable toma forma.
        - Se realiza la comprobación de tipos
        - Se agrega la información implícita (variables no declaradas)
        - Se agrega a la tabla de símbolos los descriptores de tipos, etc. a la vez que se hacen consultas para realizar comprobaciones.
        - Se hacen las comprobaciones de nombres. Ej: toda variable debe estar declarada.
        - Es el nexo entre el análisis y la síntesis.
    - Síntesis:

- En esta etapa se construye el programa ejecutable.
- Se genera el código necesario y se optimiza el programa generado.
- Si hay traducción separada de modulos, es en esta etapa cuando se linkedita.
- Se realiza el proceso de optimización. Optativo.
- Optimización del código
- Generación del código intermedio:
    - Características de esta representación:
        - Debe ser fácil de producir
        - Debe ser fácil de traducir al programa objeto.

Ejemplo: un formato de código intermedio es el código de tres direcciones.
Forma: A:=B op C, donde A,B,C son operandos y op es un operador binario.
Se permiten condiciones simples y saltos.
while (a >0) and (b<(a*4-5)) do a:=b*a-10;

```
L1: if (a>0) goto L2
        goto L3
L2: t1:=a*4
        t2:=t1-5
        if (b < t2) goto L4
        goto L3
```

```
L4: t1:=b*a
        t2:=t1-10
        a:=t2
        goto L1
        L3: …….
```



SEMANTICA OPERACIONAL
VARIABLE

Semántica de los lenguajes de programación

ENTIDAD	ATRIBUTO
- Variable	- nombre, tipo, área de memoria, etc.
- Rutina	- nombre, parámetros formales, parámetros reales, etc.
- Sentencia	-acción asociada

DESCRIPTOR: lugar donde se almacenan los atributos.

Concepto de ligadura (BINDING)

Los programas trabajan con **entidades**

↓

Las entidades tienen **atributos**

↓

Estos atributos tienen que establecerse antes de poder usar la entidad

↓

Ligadura: es la asociación entre la entidad y el atributo

Ligadura

Diferencia entre los lenguajes de programación.

- El número de **entidades.**
- El número de **atributos** que se les pueden ligar.
- El **momento** en que se hacen las ligaduras (**binding time**).
- La **estabilidad** de la ligadura: una vez establecida ¿se puede modificar?

Momento de ligadura

- Definición del lenguaje → estático
- Implementación del lenguaje → estático
- Compilación (procesamiento) → estático
- Ejecución → dinámico

Momento y estabilidad

- Una **ligadura es estática** si se establece antes de la ejecución y no se puede cambiar. El termino estático referencia al momento del binding y a su estabilidad.
- Una **ligadura es dinámica** si se establece en el momento de la ejecución y puede cambiarse de acuerdo a alguna regla específica del lenguaje.
    - Excepción: constantes.
- Ejemplos:
    - En definición:
        - Forma de las sentencias
        - Estructura del programa
        - Nombres de los tipos predefinidos
    - En implementación:
        - Representación de los números y sus operaciones
    - En compilación:
        - Asignación del tipo a las variables
    - En ejecución:
        - Variables con sus valores
        - Variables con su lugar de almacenamiento

Concepto

- Memoria principal: celdas elementales, identificadas por una dirección.
- El contenido de una celda es una representación codificada de un valor.
- <NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>
    - Nombre: string de caracteres que se usa para referenciar a la variable (identificador).
    - Alcance: es el rango de instrucciones en el que se conoce el nombre
    - Tipo: valores y operaciones
    - L-Value: es el lugar de memoria asociado con la variable (tiempo de vida).
    - R-Value: es el valor codificado almacenado en la ubicación de la variable

- <**NOMBRE**, ALCANCE, TIPO, L-VALUE, R-VALUE>
    - Aspectos de diseño:

        - Longitud máxima. Algunos ejemplos: Fortran: 6 ; Python: sin límite ; C: depende del compilador, suele ser de 32 y se ignora si hay más.
    - Caracteres aceptados (conectores). Ejemplo: Python, C, Pascal: _ ; Ruby: solo letras minúsculas para variables locales.
    - Sensitivos: SUM = sum = SUM?. Ejemplo: C y Python sensibles a mayúsculas y minúsculas. Pascal no sensible a mayúsculas y minúsculas.
    - Palabra reservada – palabra clave

- <NOMBRE, **ALCANCE**, TIPO, L-VALUE, R-VALUE>
    - El alcance de una variable es el rango de instrucciones en el que se conoce el nombre (visibilidad)
    - Las instrucciones del programa pueden manipular una variable a través de su nombre dentro de su alcance.
    - Los diferentes lenguajes adoptan diferentes reglas para ligar un nombre a su alcance.
    - Alcance estático:
        - Llamado alcance léxico
        - Define el alcance en términos de la estructura léxica del programa.
        - Puede ligarse estáticamente a una declaración (explícita o implícita) examinando el texto del programa, sin necesidad de ejecutarlo.
        - La mayoría de los lenguajes adoptan reglas de ligadura de alcance estático.
    - Alcance dinámico:
        - Define el alcance del nombre de la variable en términos de la ejecución del programa.
        - Cada declaración de variable extiende su efecto sobre todas las instrucciones ejecutadas posteriormente, hasta que una nueva declaración para una variable con el mismo nombre es encontrado durante la ejecución.
        - APL, Lisp (original), Afnix (llamado Aleph hasta el 2003), Tc (Tool Command Language), Perl.

- <NOMBRE, ALCANCE, **TIPO**, L-VALUE, R-VALUE>
    - Definición:
        - Conjunto de valores
        - Conjunto de operaciones
    - Antes de que una variable puede ser referenciada debe ligársele un tipo.
    - Protege a las variables de operaciones no permitidas.

    Chequeo de tipos: verifica el uso correcto de las variables
    - Predefinidos
        - Tipos base: son los que están descriptos en la definición.
            - Tipo boolean: valores (true, false) operaciones (and, or, not)
        - Los valores se ligan en la implementación a representación de máquina.
    - Definidos por el usuario
        - Constructores
        - Los lenguajes permiten al programador mediante la declaración de tipos definir nuevos tipos a partir de los predefinidos y los constructores.
    - TADs:
        - No hay ligadura por defecto, el programador debe especificar la representación y las operaciones.
        - Estructura de datos que representan al nuevo tipo.
        - Rutinas usadas para manipular los objetos de este nuevo tipo.
    - ESTATICO VS DINAMICO
        - Las reglas dinámicas son más fáciles de implementar.
        - Son menos claras en cuanto a disciplina de programación.
        - El código se hacen más difícil de leer.

        - Conceptos asociados con el alcance

- **Local:** son todas las referencias que se han creado dentro del programa o subprograma.
- **No local:** son todas las referencias que se utilizan dentro del subprograma pero que no han sido creadas en él.
- **Global:** son todas las referencias creadas en el programa principal.

- Espacios de nombres
  - Definición: un espacio de nombre es una zona separada donde se pueden declarar y definir objetos, funciones y en general, cualquier identificador de tipo, clase, estructura, etc.; al que se asigna un nombre o identificador propio.
  - Utilidad: ayudan a evitar problemas con identificadores con el mismo nombre en grandes proyectos o cuando se usan bibliotecas externas.
- Momentos – **estático**
  - El tipo se liga en compilación y no puede ser cambiado.
    - El chequeo de tipo también será estático
      - Estático → explicito, implícito, inferido. (Pascal, Algol, Simula, ADA, C, C++. Java, etc)
- Momento – estático – **explicito**
  - La ligadura se estable mediante una declaración.
- Momento – estático – **implícito**
  - La ligadura se deduce por las regalas.
  - Ej. Fortran:
    - Si el nombre comienza con I a N es entera.
    - Si el nombre comienza con la letra A-H o O-Z es real.

  Semánticamente la explicita y la implícita son equivalentes, con respecto al tipado de las variables, ambos son estáticos. El momento en que se hace la ligadura y su estabilidad es el mismo en los dos lenguajes.
- Momento – estático – **inferido**
  - El tipo de una expresión se deduce de los tipos de sus componentes.
  - Lenguaje funcional. Ej. Lisp

    Si se tiene en un script → doble x = 2 * x

    Su no está definido el tipo se infiere → doble :: num -> num
- Momento – **Dinámico**
  - El tipo se liga en ejecución y puede cambiarse
    - Más flexible: programación genérica
    - Más costoso en ejecución: mantenimiento de descriptores.
    - Variables polimórficas
    - Chequeo dinámico
    - Menor legibilidad

- <NOMBRE, ALCANCE, TIPO, **L-VALUE**, R-VALUE>
  - Área de memoria ligada a la variable
  - Tiempo de vida (lifetime) o extensión:
    - Periodo de tiempo que exista la ligadura
  - Alocación:
    - Momento que se reserva la memoria

  El tiempo de vida es el tiempo en que la variable este alocada en memoria
  - Momentos – **Alocación**
    - Estática: sensible a la historia
    - Dinámica:
      - Automática; cuando aparece la declaración
      - Explicita: a través de algún constructor.

        - Persistente: su tiempo de vida no depende de la ejecución:
            - Existe en el ambiente
            - Archivos – bases de datos

- <NOMBRE, ALCANCE, TIPO, L-VALUE, **R-VALUE**>
    - Valor almacenado en el l-valor de la variable
    - Se interpreta de acuerdo al tipo de la variable
    - Objeto: (l-valor, r-valor)
        - X := x + 1 → 1° x es L-valor ; 2° x es R-valor
    - Momentos:
        - Dinámico: por naturaleza
            - B := A se copia el r-valor de a en el l-valor de b
            - A:=17
        - Constantes: se congela el valor
    - Inicialización:
        - ¿Cuál es el r-valor luego de crearse la variable?
            - Ignorar el problema: lo que haya en memoria
            - Estrategia de inicialización:
                - Inicialización por defecto:
                    - Enteros se inicializan en 0, los caracteres en blanco, etc.
                    - Inicialización en la declaración.

VARIABLES ANONIMAS Y REFERENCIAS

- Algunos lenguajes permiten que el r-valor de una variable sea una referencia al l-valor de otra variable.

ALIAS

- Dos variables comparten un objeto si sus caminos de acceso conducen al objeto. Un objeto compartido modificado vía un camino, se modifica para todos los caminos.
- Alias: dos nombres que denotan la misma entidad en el mismo punto de un programa.
    - Distintos nombres → 1 entidad
- Dos variables son alias si comparten el mismo objeto de dato en el mismo ambiente de referencia. El uso de alias puede llevar a programas de difícil lectura y a errores.
- Efecto lateral: modificación de una variable no local.

CONCEPTO DE SOBRECARGA

- Sobrecarga:
    - 1 nombre → distintas entidades
    - Sobrecarga: un nombre esta sobrecargado si:
        - En un momento, referencia más de una entidad
        - Hay suficiente información para permitir establecer la ligadura unívocamente.

SEMANTICA OPERACIONAL
UNIDADES DE PROGRAMA

- Unidades:
    - Los lenguajes de programación permiten que un programa este compuesto por unidades.
      UNIDAD → acción abstracta
    - En general se las llama **rutinas**
        - ➔ Procedimientos
        - ➔ Funciones -> un valor
    - Analizaremos las características sintácticas y semánticas de las rutinas y los mecanismos que controlan el flujo de ejecución entre rutinas con todas las ligaduras involucradas.

  Hay lenguajes que SOLO tienen “funciones” y “simulan” los procedimientos con “sunciones que devuelven void”. Ej: C, C++, Python, etc.

- <**NOMBRE**, ALCANCE, TIPO, L-VALUE, R-VALUE>
  - String de caracteres que se usa para invocar a la rutina (identificador)
  - El nombre de la rutina se introduce en su declaración.
  - El nombre de la rutina es lo que se usa para invocarlas.
- <NOMBRE, **ALCANCE**, TIPO, L-VALUE, R-VALUE>
  - Rango de instrucciones donde se conoce su nombre.
    - El alcance se extiende desde el punto de su declaración hasta algún constructor de cierre.
    - Según el lenguaje puede ser estático o dinámico.
  - Activación: la llamada puede estar solo dentro del alcance de la rutina.
- DEFINICION VS DECLARACION
  - Algunos lenguajes (C, C++, Ada, etc.) hacen distinción entre definición y declaración de las rutinas.
  - Si el lenguaje distingue entre la declaración y la definición de una rutina permite manejar esquemas de rutinas mutuamente recursivas.
- <NOMBRE, ALCANCE, **TIPO**, L-VALUE, R-VALUE>
  - El encabezado de la rutina define el **tipo de los parámetros** y el **tipo del valor de retorno** (si lo hay).
  - **Signatura:** permite especificar el tipo de una rutina.
    Una rutina *fun* que tiene como entrada parámetros y tipo T1, T2, Tn y devuelve un valor de tipo R, puede especificarse con la siguiente signatura.
    - *Fun*: T1xT2…Tn → R
  - Un llamado a una rutina es correcto si está de acuerdo con el tipo de la rutina.
  - La conformidad requiere la correspondencia de tipos entre parámetros formales y reales.
- <NOMBRE, ALCANCE, TIPO, **L-VALUE, R-VALUE**>
  - L-VALUE: es el lugar de memoria en el que se almacena el cuerpo de la rutina.
  - R-VALUE: la llamada a la rutina causa la ejecución de su código, eso constituye su r-valor.
    - Estático: el caso más usual.
    - Dinámica: variables de tipo rutina.

    Se implementan a través de punteros a rutinas.
- Representación en ejecución:
  - La definición de la rutina especifica un proceso de computo.
  - Cuando se invoca una rutina se ejecuta una instancia del proceso con los particulares valores de los parámetros.
  - **Instancia de la unidad:** es la representación de la rutina en ejecución:
    - **Segmento de código:** instrucciones de la unidad se almacenan en la memoria de instrucción C → contenido fijo.
    - **Registro de activación:** datos locales de la unidad se almacenan en la memoria de datos D → contenido cambiante.
- Procesador abstracto – utilidad:
  - El procesador nos servirá para comprender que efecto causan las instrucciones del lenguaje al ser ejecutadas.
  - Semántica intuitiva.
  - Se describe la semántica de lenguaje de programación a través de reglas de cada constructor del lenguaje traduciéndolo en una secuencia de instrucciones equivalentes del procesador abstracto.
  - SIMPLESEM:
    - Memoria de código: C(y) valor almacenado en la yesima celda de la memoria de código. Comienza en ceo.
    - Memoria de datos: D(y) valor almacenado en la yesima celda de la memoria de datos. Comienza en cero.
      "y" representa el l-valor, D(y) o C(y) su r-valor.
    - Ip: puntero a la instrucción que se está ejecutando.
      - Se inicializa en cero y en cada ejecución se actualiza cuando se ejecuta cada instrucción.
      - Direcciones de C.

Ejecución:

- Obtener la instrucción actual para ser ejecutada (C[ip])
- Incrementar ip
- Ejecutar la instrucción actual

- Procesador abstracto – Instrucciones
  - SET: setea valores en la memoria de datos *set target, source*
    - Copia el valor representado por source en la dirección representada por target
    - Ejemplo: *set 10,D[20]* copia el valor almacenado en la pos 20, en la pos 10.
  - JUMP: bifurcación incondicional.
    - *Jump 47*, la próxima instrucción a ejecutarse será la que este almacenada en la dirección 41 de C
  - JUMPT: bifurcación condicional, bifurca si la expresión se evalua como verdadera.
    - *Jumpt 47, D[13]>D[8]* bifurca si el valor almacenado en la celda 13 es mayor que el almacenado en la celda 8
- Elementos en ejecución:
  - Punto de retorno: es una pieza cambiante de información que debe ser salvada en el registro de activación de la unidad llamada.
  - Ambiente de referencia:
    - Ambiente local: variables locales, ligadas a los objetos almacenados en su registro de activación.
    - Ambiente no local: variables no locales, ligadas a objetos almacenados en los registros de activación de otras unidades.
- ESTRUCTURA ED EJECUCION DE LOS LENGUAJES DE PROGRAMACION
  - **Estático: espacio fijo**
    - El espacio necesario para la ejecución se deduce del código
    - Todo los requerimientos de memoria necesarios se conocen antes de la ejecución
    - La Alocacion puede hacerse estáticamente
    - No puede haber recursión
  - **Basado en pila: espacio predecible**
    - El espacio se deduce del código. Algol-60
    - Programas más potentes cuyos requerimientos de memoria no puede calcularse en traducción.
    - La memoria a utilizarse es predecible y sigue una disciplina last-in-first-out.
    - Las variables se alocan automáticamente y se alocan cuando el alcance se termina
    - Se utiliza una estructura de pila para modelizarlo.
    - Una pila no es parte de la semántica del lenguaje, es parte de nuestro modelo semántico.
  - **Dinámico: espacio impredecible**
    - Lenguajes con impredecible uso de memoria.
    - Los datos son alocados dinámicamente solo cuando se los necesita durante la ejecución.
    - No pueden modelizarse con una pila, el programador puede crear objetos de dato en cualquier punto arbitrario durante la ejecución del programa.
    - Los datos se alocan en la zona de memoria heap.
- C1: LENGUAJE SIMPLE
  - Sentencias simples
  - Tipos simples
  - Sin funciones
  - Datos estáticos de tamaño fijo
  - Un programa = una rutina main ()
    - Declaraciones
    - Sentencias
  - E/S: get/print

- C2: C1 + RUTINAS INTERNAS
    - Programa =
        - Datos globales
        - Declaraciones de rutinas
        - Rutina principal
    - Rutinas internas
        - Disjuntas: no pueden estar anidadas
        - No son recursivas
    - Ambiente de las rutinas internas
        - Datos locales
        - Datos globales

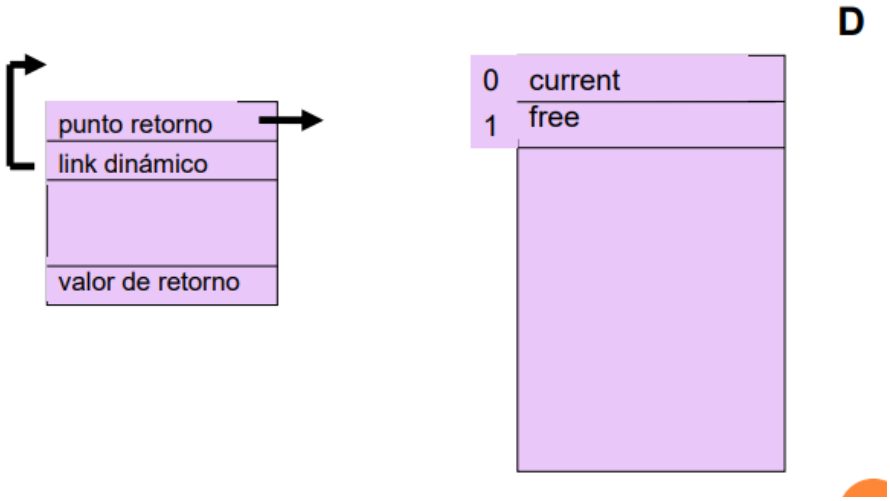
- C2'
    - El compilador no puede ligar variables locales a direcciones absolutas
    - Tampoco variables globales
    - Linkeditor:
        - Encargado de combinar los módulos
        - Ligar la información faltante
    - C2 y C2' no difieren semánticamente.

ESQUEMAS DE EJECUCION

CASOS DE C3 A C6

- C3: C2 + recursión y valor de retorno
    - Esquema basado en pila
    - Rutinas con capacidad de llamarse a sí mismas (recursión directa) o de llamar a otra rutina en forma recursiva (recursión indirecta).
    - Rutinas con la capacidad de devolver valores, es decir, funciones.
- C3: Funcionamiento
    - El registro de activación de cada unidad será de tamaño fijo y conocido, pero no se sabrá cuantas instancias de cada unidad se necesitaran durante la ejecución.
    - Igual que en C2 el compilador puede ligar cada variable con su desplazamiento dentro del correspondiente registro de activación. El desplazamiento es estático.
    - La dirección donde se cargara el registro de activación, es dinámica, por lo tanto, la ligadura con las direcciones absolutas en la zona de datos de la memoria, solo puede hacerse en ejecución.
    - Cada nueva invocación aloca un nuevo registro de activación y se establecen las nuevas ligaduras ante el segmento de código y el nuevo registro de activación.
    - Hay que tener en cuenta que cuando la instancia actual de la unidad termine de ejecutarse, su registro de activación no se necesitara más, por lo tanto se puede liberar el espacio ocupado por su registro de activación y dejar el espacio disponible para nuevos registros
    - Las unidades puede devolver valores (funciones) y esos valores no deberían perderse cuando se desactive la unidad.
- C3: Datos necesarios
    - Para manejar la Alocacion dinámica necesitamos nuevos elementos:
        - Valor de retorno: al terminar una rutina se desaloca su RA, por lo tanto la rutina llamante debe guardar en su RA el valor de retorno de la rutina llamada.
        - Link dinámico: contiene un puntero a la dirección base del registro de activación de la rutina llamadora.
        - Current: dirección base del registro de activación de la unidad que se esté ejecutando actualmente
        - Free: próxima dirección libre en la pile
        - Cadena dinámica: cadena de links dinámicos originada en la secuencia de registros de activación activos. Representa la secuencia dinámica de unidades activadas.

- C3: Moldes



- C4: Estructura de bloque
  - C4' permite que dentro de las sentencias compuestas aparezcan declaraciones locales
  - C4'' permite la definición de una rutina dentro de otras rutinas (anidamiento de rutinas)
  - Estas características conforman el concepto de estructura en bloque:
    - Controla el alcance de las variables
    - Define el tiempo de vida de las variables
    - Divide el programa en unidades más pequeñas.
  - Los bloques pueden ser:
    - Disjuntos (no tiene porción común)
    - Anidados (un bloque está completamente contenido en otro)
- C4': Anidamiento vía sentencias compuestas
  - Un bloque tiene forma de una sentencia compuesta: {<lista de declaraciones>;<lista de sentencias>}
  - Las variables tienen alcance local: son visibles dentro de la sentencia compuesta, incluyendo cualquier sentencia compuesta anidada en ella
  - Si en el anidamiento, hay una nueva declaración de un nombre, la declaración interna enmascara la externa del mismo nombre.
- C4': Sentencias compuestas
  - Alocacion: implementación
    - Estático: incluir las necesidades dentro del registro de activación de la unidad a la que pertenece. Simple y eficiente en tiempo.
    - Dinámico: alocar el espacio dinámicamente cuando se ejecutan las sentencias. Eficiente en espacio.
- C4'': Acceso al ambiente no local
  - Link estático: apunta al registro de activación de la unidad que estáticamente la contiene.
  - La secuencia de links estáticos se denomina cadena estática.
- C5: Datos más dinámicos
  - C5': registro de activación cuyo tamaño se conoce cuando se activa la unidad. (Datos semidinamicos)
  - C5'': los datos pueden alocarse durante la ejecución. (Datos dinámicos)
- C5': Datos semidinamicos
  - Variables cuyo tamaño se conoce en compilación.
  - Arreglos dinámicos:
    - *Type VECTOR is array (INTEGER range <>);* define un arreglo con índice irrestricto
      A: VECTOR (0..N);
      B: VECTOR (1..M);
      N y M deben ligarse a algún valor entero para que A y B puedan alocarse en ejecución (referencia al ambiente no local o parámetros)

- C5': Implementación de arreglos dinámicos.
    - Compilación: se reserva lugar en el registro de activación para los descriptores de los arreglos dinámicos.
    - Todos los accesos al arreglo dinámico son traducidos como referencias indirectas a través del puntero en el descriptor, cuyo desplazamiento se determina estáticamente.
- C5': Datos semidinamicos
    - Ejecución: el registro de activación se aloca en varios pasos:
        1. Se aloca el almacenamiento para los datos de tamaño conocido estáticamente y para los descriptores de los arreglos dinámicos.
        2. Con la declaración se calculan las dimensiones en los descriptores y se extiende el registro de activación para incluir el espacio para la variable dinámica.
        3. Se fija el puntero del descriptor con la dirección del área alocada.
- C5'': Datos dinámicos
    - Se aloca explícitamente durante la ejecución mediante instrucciones de alocación.
    - El tiempo de vida no depende de la sentencia de alocación, vivirá mientras este apuntada.
- C6: Lenguajes dinámicos
    - Se trata de aquellos lenguajes que adoptan más reglas dinámicas que estáticas.
    - Usan tipado dinámico y reglas de alcance dinámicas.
    - Se podrían tener reglas de tipado dinámicas y de alcance estático, pero en la práctica las propiedades dinámicas se adoptan juntas.
    - Una propiedad dinámica significa que las ligaduras correspondientes se llevan a cabo en ejecución y no en compilación.
- Variables estáticas: **C1 y C2** (estático)
- Variables semiestaticas o automáticas: **C3 y C4** (pila)
- Variables semidinamicas: **C5'** (pila)
- Variables dinámicas: **C5''** (heap)
- Tipos y alcance dinámico: **C6** (heap)

RUTINAS

- Conjunto de sentencias que representan acción abstracta.
- Representan una unidad de programa.
- Amplían el lenguaje
- El ejemplo más usual y útil presente desdelo primeros lenguajes ensambladores son las subprogramas, unidades de programa con llamada explicita, que responden al esquema de call/return.
- A nivel de diseño permite definir una operación creada por el usuario a semejanza de las operaciones primarias integradas en el lenguaje.
- Formas de subprogramas:
    - Procedimientos:
        - Definen nuevas sentencias creadas por el usuario.
        - Los resultados los produce en variables no locales o en parámetros que cambian su valor.
        - En Fortran se los llaman subrutinas, pero en general se los llama procedimientos.
    - Funciones:
        - Define un nuevo operador.
        - Similar a las funciones matemáticas ya que solo producen un valor y no producen efectos laterales.
        - Se las invoca dentro de expresiones con su lista de parámetros reales.
        - El valor que produce reemplaza a la invocación dentro de la expresión.
- Subprogramas:
    - Definición:
        - Especificación o encabezado: contiene el nombre y la lista de parámetros.
        - Implementación: contiene la definición de las variables locales y las sentencias que forman su código.

- Uso:
  - El subprograma se lo invoca por su nombre, indicando la lista de argumentos que se usaran y generando una instancia de la unidad.
- Conclusiones:
  - Al diseñar un subprograma el programador se concentra en cómo trabaja dicho subprograma.
  - Cuando se usa el subprograma se ignorara el cómo y se concentra en el que se puede hacer. Es decir la implementación permanece oculta.
  - Con una sola definición se pueden crear muchas activaciones. La definición de un subprograma es un patrón para crear activaciones durante la ejecución.
  - Un subprograma es la implementación de una acción abstracta y su invocación representa el uso de dicha abstracción.
  - Codificar un subprograma es como si hubiéramos incorporado una nueva sentencia a nuestro lenguaje.

PARAMETROS

- Formas de compartir datos entre diferentes unidades:
  - A través del acceso al ambiente no local.
  - A través del uso de parámetros.
- A través del acceso al ambiente no local:
  - Ambiente común explícito:
    - Ejemplos:
      - COMMON de FORTRAN
      - Con uso de paquetes de ADA
      - Con variables externas de PL/1
  - Ambiente no local implícito
    - Utilizando regla de alcance dinámico.
    - Utilizando regla de alcance estático.
- Pasaje de parámetros:
  - El pasaje de parámetros es el más flexible y permite la transferencia de diferentes datos en cada llamada.
  - Proporciona ventajas en legibilidad y modificabilidad.
  - Nos permiten compartir los datos en forma abstracta ya que indican con precisión qué es exactamente lo que se comparte.
  - Lista de parámetros:
    - Conjunto de datos que se van a compartir
      - Parámetros **reales**:
        - Parámetros que se codifican en la invocación del subprograma.
        - Puede ser local a la unidad llamadora, o ser a su vez un parámetro formal de ella o un dato no local pero visible en dicha unidad o una expresión.
      - Parámetros **formales**:
        - Parámetros declarados en la especificación del subprograma.
        - Contiene los nombres y los tipos de los datos compartidos.
        - En general son similares a variables locales.

      Si es una función debe darse el tipo de lo que se retorna.

  - Evaluación de los parámetros reales y ligadura con los parámetros formales:
    - Evaluación:
      - En general en el momento de la invocación primero se evalúa los parámetros reales, y luego se hace la ligadura antes de transferir el control a la unidad llamada.
    - Ligadura:
      - Posicional: se corresponden con la posición que ocupan en la lista.

- Palabra clave o nombre: se corresponden con el nombre por lo tanto pueden estar colocados indistintamente en la lista.

  En Ada pueden mezclarse ambos métodos. En C++, Ada, Python los parámetros formales pueden tener valores por defecto, con lo cual a veces no es necesario listarlos todos en la invocación.

- Clases de parámetros: datos y subprograma.
  - Parámetros datos: hay diferentes formas de transmitir los parámetros hacia y desde el programa llamado. Desde el punto de vista semántico los parámetros formales pueden ser:
    - Modo IN: el parámetro formar recibe el dato desde el parámetro real.
    - Modo OUT: el parámetro formar envía el dato al parámetro real.
    - Modo IN/OUT: el parámetro formar recibe el dato del parámetro real y el parámetro formal le envía el dato al parámetro real.
  - **Modo IN**: puede ser por valor o por valor constante.
    - Por valor:
      - El valor del parámetro real se usa para inicializar el correspondiente parámetro real al invocar la unidad.
      - Se transfiere el dato real.
      - En este caso el parámetro formal actúa como una variable local de la unidad llamada.

      Desventaja: consume el tiempo para hacer la copia y el almacenamiento para duplicar el dato.

      Ventaja: protege los datos de la unidad llamadora, el parámetro real no se modifica.

    - Por valor constante:
      - No indica si se realiza o no la copia, lo que estable es que la implementación debe verificar que el parámetro real no sea modificado.
      - Ejemplo: parámetros IN de ADA.

      Desventaja: requiere realizar más trabajo para implementar los controles.

      Ventaja: protege los datos de la unidad llamadora, el parámetro real no se modifica.

  - **Modo OUT**: puede ser por resultado o por resultado de funciones.
    - 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 local, sin valor inicial.

      Desventaja: consume tiempo y espacio. Si se repiten los parámetros reales los resultados pueden ser diferentes. Se debe tener en cuenta el momento en que se evalúa el parámetro real.

      Ventajas: protege los datos de la unidad llamadora, el parámetro real no se modifica en la ejecución de la unidad llamada.

    - Por resultado de funciones:
      - El resultado de una función puede devolverse con el return como en Python, C, etc., o como en Pascal en el nombre de la función (ultimo valor asignado) que se considera como una variable local.
      - Dicho resultado reemplaza la invocación en la expresión que contiene el llamado.
  - **Modo IN/OUT**: puede ser por valor-resultado, por referencia o por nombre.

- 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 se 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.
    - Cada referencia al parámetro formal es una referencia local.
    - Tiene las desventajas y las ventajas de ambos.
- Por referencia:
    - Se transfiere la direccion del parámetro real al parámetro formal.
    - El parámetro formal será una variable local a la unidad llamadora que contiene la direccion en el ambiente no local.
    - Cada referencia al parámetro forma será a un ambiente no local. Esto significa que cualquier cambio que se realice en el parámetro formal dentro del cuerpo del subprograma quedara registrado en el parámetro real.
    - El parámetro real es compartido por la unidad llamada.

      Desventaja: el acceso al dato es más lento por la indireccion. Se pueden modificar el parámetro real inadvertidamente. Se pueden generar alias y estos afectan la legibilidad y por lo tanto la confiabilidad, hacen muy difícil la verificación de programas.

      Ventaja: eficiente en tiempo y espacio. No se realizan copias del dato.

- Por nombre:
    - El parámetro formal es sustituido textualmente por el parámetro real. Es decir se establece la ligadura entre parámetro formal y parámetro real en el momento de la invocación pera la ligadura de valor se difiere hasta el momento en que se lo utiliza.
    - El objetivo es otorgar mayor flexibilidad a través de esta evolución de valor diferida.
    - Si el dato a compartir es:
        - Un único valor se comporta exactamente igual que el pasaje por referencias.
        - Si es una constante es equivalente a por valor.
        - Si es un elemento de un arreglo puede cambiar el suscripto entre las distintas referencias.
        - Si es una expresión se avalúa cada vez.
    - Para implementarlo se utilizan los thunks que son procedimientos sin nombre. Cada aparición del parámetro formal se reemplaza en el cuerpo de la unidad llamado por una invocación a un thunks que en el momento de la ejecución activara al procedimiento que evaluara el parámetro real en el ambiente apropiado.
    - Es un método más flexible pero más lento, ya que debe evaluarse cada que se lo usa.
    - Es difícil de implementar y genera soluciones confusas para el lector y el escritor.

- Pasaje de parámetros en funciones:
    - Las funciones no deberían producir efectos laterales. Es decir no deben alterar el valor de ningún dato (local ni no local) solo producir un resultado.
    - Los parámetros formales deberían ser siempre modo IN. (no Pascal y C, si ADA).
    - Ortogonalidad. Los resultados deberían poder ser de cualquier tipo.
      Ejemplo: Ada, Pascal y Modula solo permiten escalares.

- Pasaje de parámetros en algunos lenguajes:
  - C:
    - Por valor, (si se necesita por referencia se usan punteros).
    - C, permite pasaje por valor constante, agregándole const.
  - Modula II y Pascal:
    - Por valor (por defecto)
    - Por referencia (opcional: var)
  - C++:
    - Igual que C más pasaje por referencia.
  - Java:
    - El único mecanismo contemplado es el paso por copia de valor. Pero como las variables de tipos NO primitivos son todas referencias a variables anónimas en el Heap, el paso por valor de una de estas variables constituye en realidad un paso por referencia de la variable.
  - Python:
    - Envía objetos que pueden ser "inmutables" o "mutables". Si es inmutable actuará como por valor y, si es mutable, ejemplo: listas, no se hace una copia sino que se trabaja sobre él.
  - ADA:
    - Por copia IN (por defecto)
    - Por resultado OUT.
    - IN OUT.
      - Para los tipos primitivos indica por valor-resultado.
      - Para los tipos no primitivos, datos compuestos (arreglos, registro) se hace por referencia.
    - Particularidad de ADA:
      - En las funciones solo se permite el paso por copia de valor, lo cual evita parcialmente la posibilidad de efectos laterales.

- Subprogramas como parámetro:
  - En algunas situaciones es conveniente poder manejar como parámetros los nombres de los subprogramas.
  - Ejemplo: supongamos un subprograma que acepta como parámetro una referencia o puntero a un arreglo de enteros, y cuyo efecto sea ordenarlo según un criterio de ordenación, que no siempre es el mismo.
    Seria natural que el nombre del procedimiento que ordena sea un parámetro.
    ADA no contempla los subprogramas como valores. Utiliza unidades genéricas.
    Pascal permite que una referencia a un procedimiento sea pasada a un subprograma.
    C permite pasaje de funciones como parámetros.
- Ambiente de referencia para las referencias no locales dentro del cuerpo del subprograma pasado como parámetro:
  - Debe determinarse cuál es el ambiente de referencia no local correcto para un subprograma que se ha invocado y que ha sido pasado como parámetro.
  - Hay varias opciones:
    - **Ligadura shallow o superficial**: el ambiente de referencia, es el del subprograma que tiene el parámetro formal subprograma. Ejemplo: SNOBOL.
    - **Ligadura Deep o profunda**: el ambiente es el del subprograma donde está declarado el subprograma usado como parámetro real. Se utiliza en los lenguajes con alcance estático y estructura de bloque.
    - El ambiente del subprograma donde se encuentra el llamado a la unidad que tiene un parámetro subprograma. Poco natural.

    - Los parámetros subrutinas se comportan muy diferente en lenguajes con reglas de alcance estático que dinámico.
    - La ligadura shallow o superficial no es apropiada para lenguajes con estructuras de bloques ya que se puede acceder a ambientes que estáticamente no le eran visibles.
    - Para lenguajes de alcance estático se utiliza la ligadura Deep y se necesita que en el momento de la invocación a la unidad con parámetro subprograma además del puntero al código, se indique cual debe ser su ambiente de referencia, es decir un puntero al lugar de la cadena estática correspondiente.
- Unidades genéricas
    - Una unidad genérica es una unidad que puede instanciarse con parámetros formales de distinto tipo. Como todas las unidades instanciadas tienen el mismo nombre, da la sensación que una unidad pudiera ejecutarse con parámetros de distinto tipo.
    - Por ejemplo una unidad genérica que ordena elementos de un arreglo, podrá instanciarse para elementos enteros, flotantes, etc.
    - Como pueden usarse diferentes instancias con diferentes subprogramas proveen la funcionalidad del parámetro subprograma.

SISTEMA DE TIPOS

Representa la abstracción de datos en los lenguajes de programación.

Trata con las componentes de un programa que son sujeto de computación. Está basado en las propiedades de los objetos de datos y las operaciones de dichos objetos.

- Tipos de datos.
- Encapsulamiento y abstracción.

Estable el tipo para cada valor manipulado.

Provee **mecanismos** de expresión:

- Expresar tipos intrínsecos o definir tipos nuevos.
- Asociar los tipos definidos con construcciones del lenguaje.

Define **reglas** de resolución:

- Equivalencia de tipos - ¿dos valores tienen el mismo tipo?
- Compatibilidad de tipos - ¿puedo usar el tipo en este contexto?
- Inferencia de tipos - ¿Cuál tipo se deduce del contexto?

Mientras más flexible el lenguaje, más complejo el sistema.

Conjunto de reglas que estructuran y organizan una colección de tipos.

El objetivo del sistema de tipos es lograr que los programas sean tan seguros como sea posible.

- Seguridad vs flexibilidad.
- Tipado fuerte o tipado débil
    - Se dice que el sistema de tipos es fuerte cuando especifica restricciones sobre como las operaciones que involucran valores de diferentes tipos pueden operarse.
      Ejemplo: a = 2 ; b= “2”
      Concatenar (a,b) //error de tipos
      Sumar (a,b) //error de tipos
      Concatenar (str(a),b) //retorna “22”
      Sumar (a,int(b)) //retorna 4
    - Lo contrario estable un sistema débil de tipos.
      Ejemplo: a = 2 ; b= “2”
      Concatenar (a,b) //retorna “22”
      Sumar (a,b) //retorna 4
- Especificación de un sistema de tipos
  Tipo y tiempo de chequeo
  Reglas de equivalencia y conversión
  Reglas de inferencia de tipo
  Nivel de polimorfismo del lenguaje
    - Tipo y tiempo de chequeo:

        - Tipos de ligadura:
            - Tipado estático: ligaduras en compilación.
                - Java, C
            - Tipado dinámico: ligaduras en ejecución, provoca más comprobaciones en tiempo de ejecución (no es seguro???)
                - PHP, Phyton, Ruby.
        - Tipado seguro: No es lo mismo que estático!!!

- Definiciones:
    - Tiempo de ligadura:
        - El tipado es estático si cada entidad/variable queda ligada a su tipo durante la compilación, sin necesidad de ejecutar el programa.
        - El tipado es dinámico si la ligadura de la variable/entidad se produce en tiempo de ejecución.
- Lenguajes fuertemente tipados
    - Si el lenguaje es fuertemente tipado el compilador puede garantizar la ausencia de errores de tipo en los programas (Ghezzi)
    - Un lenguaje se dice fuertemente tipado (type safety) si el sistema de tipos impone restricciones que aseguran que no se producirán errores de tipo en ejecución.
    - Un lenguaje se dice fuertemente tipado (type safety) si todos los errores de tipo se detectan.

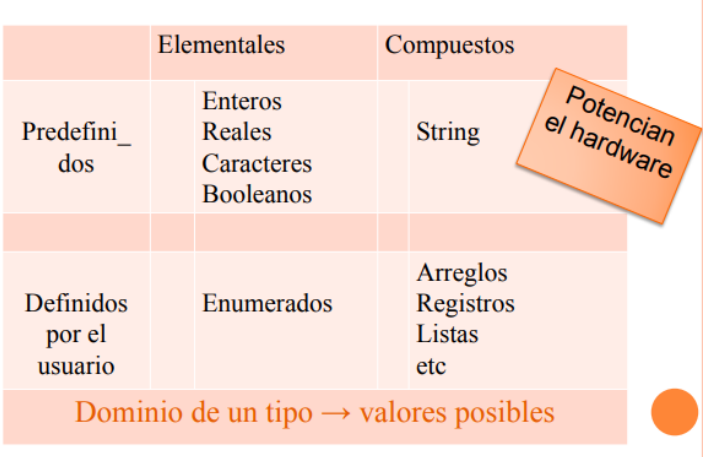
  En esta concepción, la intención es evitar los errores de aplicación y son tolerados los errores de los lenguajes (sintácticos o semánticos), detectados tan pronto como sea posible.
- ¿Qué es un tipo de dato?
    - Desde el punto de vista **denotaciones** es:
        - Conjunto de valores sobre un dominio.
    - Desde el punto de vista **constructivo** puede ser:
        - Primitivo (built-in o predefinido) provisto por el lenguaje.
        - Compuesto (composite o derivado) empleando constructores de tipos.
    - Desde el punto de vista de la **abstracción**
        - Una interfaz a una representación.
        - Conjunto de operaciones con semántica bien definida y consistente.

- Tipos de datos

  Dominio + Abstracción

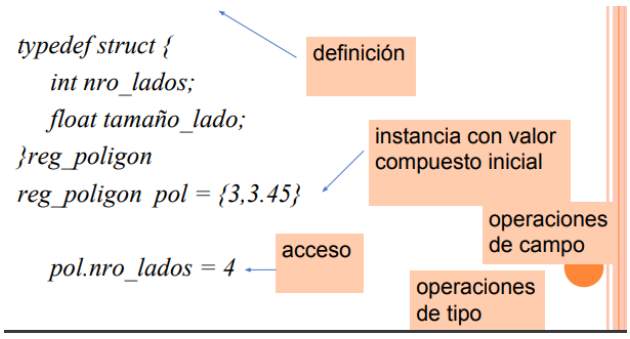
  Valores Operaciones

  Los tipos de datos “capturan la naturaleza de los datos del problema que serán manipulados por los programas”
    - Operaciones: única forma de manipular los objetos instanciados.
    - Tipo: comportamiento abstracto de un conjunto de objetos y un conjunto de operaciones.

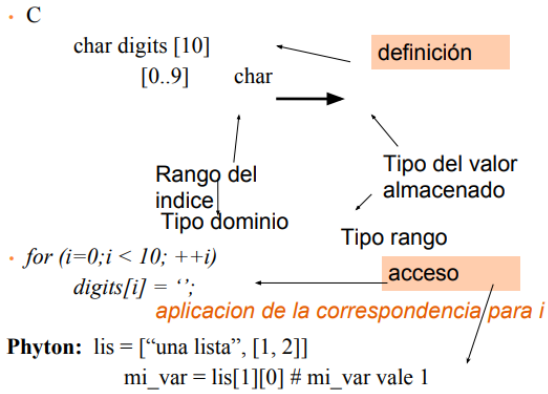


    - Nueva estructura → Tipo
        - Arreglos, registros, listas.
    - Nuevo comportamiento → TAD

        - Clases, paquetes, unidades.
    - Tipos predefinidos elementales:
        - Números: Enteros – Reales (Conversiones)
        - Caracteres: (Longitud máxima)
        - Booleanos: (Valores)
        - Ventajas:
            - Invisibilidad de la representación.
            - Verificación estática.
            - Desambiguar operadores.
            - Control de precisión.
    - Tipos definidos por el usuario:
        - Separan la especificación de la implementación. Se definen los tipos que le problema necesita.
        - Definir nuevos tipos e instanciarlos.
        - Chequeo de consistencia.
        - Legibilidad: elección apropiada de nuevos nombres.
        - Estructura jerárquica de las definiciones de tipos: proceso de refinamiento.
        - Modificabilidad: solo se cambia en la definición.
        - Factorización: se usa la cantidad de veces necesarias.
        - La instanciación de los objetos en un tipo dado implica una descripción abstracta de sus valores. Los detalles de la implementación solo quedan en la definición del tipo.
    - Tipos definidos por el usuario – enumerativos:
        - Dominio: lista de constantes simbólicas.
        - Operaciones: comparación, asignación y posición en la lista.
        - Potencian la expresividad del lenguaje.
            - Se define el nuevo tipo enumerado.
            - Se instancia el tipo
        - Noción de orden (predecesor y sucesor):
            - Relaciones de >, <, <=, >=, =
        - Ambigüedad:
    - Enumerados – Subrangos:
        - Dominio: subconjunto de un tipo entero o de un enumerado.
        - Operaciones: hereda las operaciones del tipo original.
            - La implementación es la que determina si pueden mezclarse variables.
        - Chequeo dinámico
    - Tipos compuestos: nuevos tipos definidos por el usuario usando los constructores.
        - Constructores: mecanismo para agrupar datos denominados compuestos.
            - Dato compuesto: posee nombre. Accesible a través de un mecanismo de selección. Posibilidad de manipular conjunto completo.
            - Rutinas: constructores que permiten combinar instrucciones elementales para formar un nuevo operador.
        - Compuestos:
            - <u>Producto cartesiano:</u>
                - C: estructuras



- Correspondencia finita:
  - Rutina: su definición es la regla de asociación de valores del tipo DT en valores del tipo RT.
    Definición intencional: que especifica una regla (la intención) en lugar de una asociación individual.
  - Arreglos: definición extensional, los valores de la función son explícitamente enumerados.



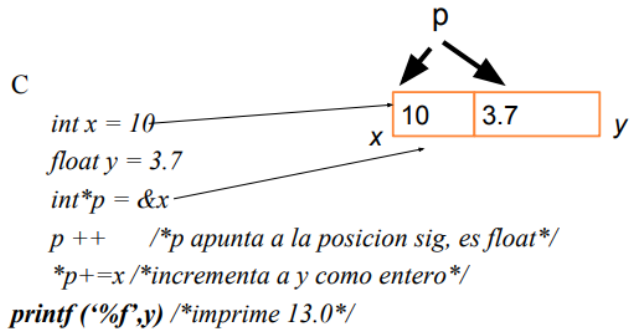
  - Limites en cada índice:
    - Ligadura en compilación: el subconjunto se fija cuando el programa se escribe. (Pascal – C)
    - Ligadura en la creación del objeto: el subconjunto se fija en ejecución, cuando se crea la instancia del objeto. (arreglos semidinamicas) (ADA)
    - Ligadura en la manipulación del objeto: es la más flexible y costosa en términos de tiempo de ejecución.
      Arreglos flexibles: los que el tamaño del subconjunto puede variar durante la vida del objeto (APL – Snobol4 – C++ - Phyton)
- Unión y Unión discriminada:
  - La unión/unión discriminada de dos o más tipos define un tipo como la disyunción de los tipos dados.
  - Permite manipular diferentes tipos en distinto momento de la ejecución.
  - Chequeo dinámico.
  - Unión discriminada:
    - Agrega un discriminante para indicar la opción elegida.
    - Si tenemos la unión discriminada de dos conjuntos S y T, y aplicamos el discriminante a un elemento e perteneciente a la unión discriminada devolverá S o T.
    - El elemento e (tipoEjemplar) debe manipularse de acuerdo al valor del discriminante.
    - En la unión y en la unión discriminada el chequeo de tipos debe hacerse en ejecución.
    - La unión discriminada se puede manejar en forma segura consultando el discriminante antes de utilizar el valor del elemento.
    - Problemas:
      - El discriminante y las variantes pueden manejarse independientemente uno de otros.
      - La implementación del lenguaje ignora los chequeos.
      - Puede omitirse el discriminante, con lo cual aunque se quisiera no se puede chequear.
- Recursión:
  - Un tipo de dato recursivo T se define como una estructura que puede contener componentes del tipo T.

- Define datos agrupados:
  - Cuyo tamaño puede crecer arbitrariamente.
  - Cuta estructura puede ser arbitrariamente compleja.
- Recursión – Implementación:
  - Los lenguajes de programación convencionales soportan la implementación de los tipos de datos recursivos a través de los punteros.
  - Los lenguajes funcionales proveen mecanismos más abstractos que enmascaran a los punteros.

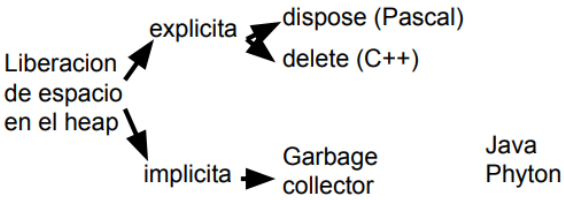
PUNTEROS

- Un puntero es una referencia a un objeto.
- Una variable puntero es una variable cuyo r-valor es una referencia a un objeto.
- Valores:
  - Direccion de memoria.
  - Valor nulo (no asignado) direccion no valida
- Operaciones: (l-valor, r-valor de la variable apuntada)
- Asignación de valor: generalmente asociado a la alocación de la variable apuntada.
- Referencias:
  - A su valor (direccion) operaciones entre punteros.
  - Al valor de la variable apuntada: de referenciación implícita.
- Inseguridad de los punteros:

1- Violación de tipos



2- Referencias sueltas – referencias dangling (punteros sueltos)
   - Si este objeto no está alocado se dice que el puntero es peligroso (dangling)
   - Una referencia suelta o dangling es un puntero que contiene una direccion de una variable dinámica que fue desalocada. Si luego se usa el puntero producirá error.
3- Liberación de memoria: objetos perdidos
   - Los objetos (apuntados) que se alocan a través de la primitiva new son alocados en la heap.
   - La memoria disponible (heap) puede agotarse.
   - Un objeto se dice accesible si alguna variable en la pila lo apunta directa o indirectamente. Un objeto es basura si no es accesible.
     - Mecanismos para desalocar memoria.
   - Si los objetos en el heap dejan de ser accesibles (objeto perdido) esa memoria podría liberarse.



- Implícita: garbage collector
  - El sistema, dinámicamente, tomara la decisión de descubrir la basura por medio de un algoritmo de recolección de basura. (LISP, ADA, Eiffel y Java, C, Phyton, Ruby)

4- Punteros no inicializados
- Peligro de acceso descontrolado a posiciones de memoria.
- Verificación dinámica de la inicialización.
- Solución:
  - Valor especial nulo: nil en Pascal
    Void en C/C++
    Null en ADA, Phyton

5- Alias

$int* p1$
$int* p2$
$int\ x$
$p1 = \&x$
$p2 = \&x$

p1 y p2 son punteros

p1 y x son alias

p2 y x también lo son

- Liberación de memoria explicita:
  - El reconocimiento de la basura recae en el programador, quien notifica al sistema cuando un objeto ya no se usa.
  - No garantiza que no haya otro puntero que apunte a esta dirección definida como basura, este puntero se transforma en dangling (puntero suelto).
    - Este error es difícil de chequear y la mayoría de los lenguajes no lo implementan por que es costoso.
- Explicita: ejemplo en C
  - Función llamada *free* libera memoria reservada de manera dinámica.
  - Puede generar referencias sueltas.
  - Para evitarlo se necesitaría una verificación dinámica para garantizar el uso correcto.

```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {

int *i;
int *p;
i = malloc(sizeof(int));
p=i;
free(i);
x=&p; //a que referencia p?
//Utilizar i despues de esto peligroso, se recomienda :
i = NULL;
return 0;
}
```

- Implícita: garbage collector
  - El sistema, durante la ejecución tomara la decisión de descubrir la basura por medio de un algoritmo de recolección de basura: garbage collector.
  - Muy importante para los lenguajes de programación que hacen un uso intensivo de variables dinámicas (LISP, Phyton)

- Eiffel y Java que uniformemente tratan a todos los objetos como referenciados por punteros proporcionan un recolector automático.
- ADA chequea dinámicamente que el tiempo de vida de los objetos apuntados sea menor igual que el del puntero.
- Garbage collector: implementaciones
  - Se ejecuta durante el procesamiento de las aplicaciones.
  - Sistema interactivo o de tiempo real: no bajar en el rendimiento y evitar los riesgos.
  - "Reference counting" este esquema supone que cada objeto en la Heap tiene un campo descriptor extra que indica la cantidad de variables que lo referencian.
    - Si es 0 → garbage

    Existen otros métodos mas complejos, pero se pierde en eficiencia.

TIPOS DE DATOS RECURSIVOS – TADs

Tipos Abstractos de Datos.

- La abstracción es el mecanismo que tenemos las personas para manejar la complejidad.
- Abstraer es representar algo descubriendo sus características esenciales y suprimiendo las que no lo son.
- El principio básico de la abstracción es la información oculta.
- TAD = Representación (datos) + Operaciones (funciones y procedimientos)
  Los tipos de datos son abstracciones y el proceso de construir nuevos tipos se llama abstracción de datos.
  Los nuevos tipos de datos definidos por el usuario se llaman *tipos abstractos de datos.*
- Los TAD satisfacen:
  - **Encapsulamiento:** la representación del tipo y las operaciones permitidas para los objetos del tipo se describe en una única unidad sintáctica.
    Refleja las abstracciones descubiertas en el diseño.
  - **Ocultamiento de la información:** la representación de los objetos y la implementación del tipo permanecen ocultos.
    Refleja los niveles de abstracción. Modificabilidad.
- Las unidades de programación de lenguajes que pueden implementar un TAD reciebn distintos nombres:
  - Modula-2 *modulo*
  - Ada *paquete*
  - C++ y Java *clase*
- Especificación de un TAD:
  - La especificación formal proporciona un conjunto de axiomas que describen el comportamiento de todas las operaciones.
  - Ha de incluir una parte de sintaxis y una parte de semántica.
  - Hay operaciones definidas por si mismas que se consideran *constructores* del TAD. Normalmente inicializan, por ejemplo:

    TAD nombre del tipo (valores que toma los datos del tipo)
    Sintaxis
    Operación (Tipo argumento, ...) → Tipo resultado
    ….
    Semántica
    Operación (valores particulares argumentos) → expresión resultado
- TAD: Clases
  - En términos prácticos, una *clase* es un tipo definido por el usuario.
  - Una clase contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones que un objeto conoce.
    - Atributos + Métodos

EXCEPCIONES

- Una excepción es una condición inesperada o inusual que surge durante la ejecución del programa y no puede ser manejada en el contexto local.
- ¿Qué hacer ante la presencia de una excepción?

- Existen tres opciones:
  - Inventar un valor que el llamador recibe en lugar de un valor valido.
  - Retornar un valor de estado al llamador, que debe verificarlo.
  - Pasar una clausura para una rutina que maneje errores.
- El manejo de excepciones por parte de los lenguajes resuelve el problema…
  - El caso normal se expresa de manera simple y directa.
  - El flujo de control se enruta a un *manejador de excepciones* solo cuando es necesario.

- Originalmente, se disponía de ejecución condicionada (PL/1) – ON condición - instrucciones
- Los lenguajes modernos ofreces bloques léxicos.
  - El bloque inicial de código para el caso normal.
  - Un bloque contiguo con el manejador de excepciones que reemplaza la ejecución del resto del bloque inicial en caso de error.
- ¿Qué debemos tener en cuenta sobre un lenguaje que provee manejo de excepciones?
  - Como **maneja** una excepción y cuál es su **ámbito:**
    - Ocurrida una excepción, que es lo que hace el lenguaje, generalmente busca el bloque de excepciones que corresponde a esa porción de código y deriva allí la ejecución.
    - Debemos tener presente cual es el alcance, generalmente igual que las variables que posee el lenguaje.
  - Como se **alcanza** una excepción:
    - **Implícita,** ocurre cuando se produce una excepción que el lenguaje tiene contemplada, como puede ser la división por cero, le lanza una excepción predefinida.
    - **Explicita,** el programador hace la invocación de la excepción a través de las instrucciones que provee el lenguaje (raise, throw, etc.).
  - Como **especificar** las unidades (manejadores de excepciones) que se han de ejecutar cuando se alcanzan las excepciones:
    - Dependiendo del lenguaje, el bloque que maneja las excepciones se debe colocar en el código en determinado lugar.
  - A donde se **cede el control** cuando se **termina** de atender las excepciones:
    - **Terminación**: se termina la ejecución de la unidad que alcanza la excepción y se transfiere el control al manejador.
    - **Reasunción**: se maneja la excepción y se devuelve el control al punto siguiente donde se invoco a la excepción, permitiendo la continuación.
  - Como se **propagan** las excepciones:
    - Debemos tener presente como hace el programa para buscar un manejador en caso de que el bloque no lo contenga explícitamente.
    - De manera general dinámicamente y puede haber combinaciones de dinámica con estática.
  - Hay excepciones **predefinidas:**
    - Algunos lenguajes que incorporan el maneja de excepciones:
    - **PL1:** continuación.
      - Fue el primer lenguaje que incorporo excepciones.
      - Utiliza el criterio de **reasunción**. Cada vez que se produce la excepción, la maneja el manejador y devuelve el control a la sentencia siguiente de donde se levantó.
      - Las excepciones son llamadas CONDITIONS.
      - Los manejadores se declaran con la sentencia ON:
        - ON CONDITIONS (nombre excepción) Manejador.
      - El manejador puede ser una instrucción o un bloque.
      - Las excepciones se alcanzan explícitamente con la palabra clave SIGNAL CONDICION (nombre excepción).
      - Este lenguaje tiene una serie de excepciones ya predefinidas con su manejador asociado. Son las **Built-in exceptions.**
        Por ejemplo: zerodivide, se levanta cuando hay una división por cero.

- A las built-in:
    - Se les puede redefinir los manejadores de la siguiente manera: ON nombre-built-in Begin .. End
    - Se las puede habilitar y deshabilitar explícitamente.
    - Se habilitan por defecto.
    - Se deshabilita anteponiendo NO nombre de la built-in al bloque, instrucción o procedimiento al que va a afectar.
- Los manejadores se ligan **dinámicamente con las excepciones.** Una excepción siempre estará ligada con el ultimo manejador definido.
- El alcance de un manejador termina cuando finaliza la ejecución de la unidad donde fue declarado.
- El alcance de un manejador de una excepción se **acota cuando se define** otro manejador para esa excepción y se **restaura cuando se desactivan** los manejadores que lo enmascararon.
- No permite que se pasen parámetros a los manejadores.
- **Desventajas:** los dos últimos puntos provocan programas difíciles de escribir y comprender y la necesidad de uso de variables globales.

▪ **ADA:** terminación.
- Criterio de **terminación.** Cada vez que se produce una excepción, se **termina el bloque** donde se levanto y se ejecuta el manejador asociado.
- Se definen en la zona de definición de las variables y tienen el mismo alcance.
- Se alcanzan explícitamente con la palabra clave **raise.**
- Los manejadores pueden encontrarse en el final de cuatro diferentes unidades de programa: **bloque, procedimiento, paquete o tarea.**
- Tiene cuatro excepciones predefinidas:
    - Constraint_error , Program_error , Storage_error , Numeric_error , Name_error , Tasking_error.
- **Propagación,** al producirse una excepción:
    - Se termina la unidad, bloque, paquete o tarea donde se alcanza la unidad.
    - Si el manejador se encuentra en ese ámbito, se ejecuta.
    - Si el manejador no se encuentra en ese lugar la excepción se propaga dinámicamente. Esto significa que se vuelve a levantar en otro ámbito.
    - Siempre tener en cuanta el alcance, puede convertirse en anónima.
- Una excepción se puede levantar nuevamente colocando solo la palabra raise.
- Tener en cuenta:
    - Es determinístico en la asociación de la excepción con el manejador.
    - Si se deseara continuar ejecutando las instrucciones de un bloque donde se lanza una excepción, es preciso “crear un bloque más interno”.

▪ **CLU:** terminación.
- Utiliza el criterio de **terminación.**
- Solamente se pueden ser **alcanzadas** por los **procedimientos.**
- Están asociadas a **sentencias.**
- Las excepciones que un procedimiento puede alcanzar se declaran en su encabezado.
- Se alcanzan explícitamente con la palabra clave **signal.**
- Los manejadores se colocan al lado de una sentencia simple o compleja. Forma de definirlos:

```
<sentencia> except
When nombre-excepcion: manejador1;
When nombre-excepcion: manejador2;
…..
When others: manejadorN;
```

End;

- Posee excepciones predefinidas con su manejador asociado. Por ejemplo, *failure*
- Se puede pasar parámetros a los manejadores.
- Una excepción se puede volver a levantar una sola vez utilizando **resignal.**
- Una excepción se puede levantar en cualquier lugar del código.
- Propagación, al producirse una excepción:
    - Se termina el procedimiento onde se levantó la excepción y devuelve el control al llamante inmediato donde se debe encontrar el manejador.
    - Si el manejador se encuentra en se ámbito, se ejecuta y luego se pasa el control a la sentencia siguiente a la que esté ligado dicho manejador.
    - Si el manejador no se encuentra en ese lugar la excepción se propaga estáticamente en las sentencias asociadas. Esto significa que le proceso se repite para las sentencias incluidas estáticamente.
    - En caso de no encuentra ningún manejador en el procedimiento que hizo la llamada se levanta una excepción **failure** y devuelve el control, terminando todo el programa.

- **C++:** terminación.
    - Utiliza el criterio de **terminación.**
    - Las excepciones pueden alcanzarse explícitamente a través de la sentencia **throw.**
    - Posee excepciones predefinidas.
    - Los manejadores van **asociados a bloques.**
    - Los bloques que puede llegar a levantar excepciones van procedidos por la palabra clave **try** y al finalizar el bloque se detallan los manejadores utilizando la palabra clave **catch (nombre de la excepción).**
    - Al levantarse una excepción dentro del bloque **try el control se transfiere** al manejador correspondiente.
    - Al finalizar la ejecución del manejador la ejecución continua como si la unidad que provoco la excepción fue ejecutara normalmente.
    - Permite pasar **parámetros al levantar la excepción:**
        - Ejemplo: throw (ayuda msg); *se está levantando la excepción ayuda y se le pasa el parámetro msg.*
    - Las rutinas en su interface pueden listar las excepciones que ellas pueden alcanzar: void rutina () throw (ayuda, zerodivide);
    - Que sucede si la rutina alcanzo otra excepción que no es contemplada en el listado de la interface:
        - En este caso NO se propaga la excepción y una función especial se ejecuta automáticamente: unexpected(), que generalmente causa abort(), que provoca el final de programa. Unexpected puede ser redefinida por el programador.
    - Que sucede si la rutina coloco en su interface el listado de posibles excepciones a alcanzar:
        - En este caso SI se propaga la excepción. Si una excepción es repetidamente propagada y no machea con ningún manejador, entonces una función terminate() es ejecutada automáticamente.
    - Que sucede si la rutina coloco en su interface una lista vacía (throw()):
        - Significa que NINGUNA excepción será propagada.
- **JAVA:** terminación.
    - Al igual que C++ las excepciones son objetos que pueden ser alcanzados y manejador por manejadores adicionados al bloque donde se produjo la excepción.
    - Cada excepción esta representada por una instancia de la clase **throwable** o de una de sus subclases (error y exception)

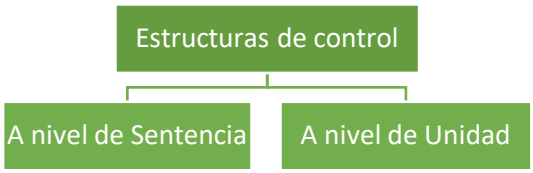
- La gestión de excepciones se lleva a cabo mediante cinco palabras claves: **try, catch, throw, throws, finally.**
- Se debe especificar mediante la cláusula **throws** cualquier excepción que se envía desde un método.
- Se debe poner cualquier código que el programador desee que se ejecute siempre, en el método **finally.**
- Fases del tratamiento de excepciones:
    - Detectar e informar el error:
        - Lanzamiento de excepciones → throw
        - Un método detecta una condición anormal que le impide continuar con su ejecución y finaliza "lanzando" un objeto excepción.
    - Recoger el error y tratarlo:
        - Captura de excepciones → bloque try-catch
        - Un método recibe un objeto excepción que le indica que otro método no ha terminado correctamente su ejecución y decide actuar en función del tipo de error.

- **Phyton:** terminación.
    - Se manejan a través de bloques try except
    - La declaración try funciona de la siguiente manera:
        - Primero, se ejecuta el bloque try (el código entre las declaraciones try y except)
        - Si no ocurre ninguna excepción, el bloque except se saltea y termina la ejecución de la declaración try.
        - Si ocurre una excepción durante la ejecución del bloque try, el resto del bloque se saltea. Luego, si su tipo coincide con la excepción nombrada luego de la palabra reservada except, se ejecuta el bloque except, y la ejecución continua luego de la declaración try.
        - Si ocurre una excepción que no coincide con la excepción nombrada en el except, esta se pasa a declaraciones try de mas afuera; si no se encuentra nada que la maneje, es una excepción no manejada y la ejecución se frena con un mensaje.
    - ¿Qué sucede cuando una excepción no encuentra un manejador en su bloque "try except"?
        - Busca estáticamente: analiza si ese try este contenido dentro de otro y si ese otro tiene un manejador para esa excepción, sino
        - Busca dinámicamente: analiza quien lo llamo y busca allí.
        - Si no se encuentra un manejador, se corta el proceso y larga el mensaje estándar de error.
        - Levanta excepciones explícitamente con "raise".
- **PHP:** terminación.
    - Modelo de **terminación.**
    - Una excepción puede ser lanzada (thrown), y atrapada ("catchet").
    - El código esta dentro de un bloque try.
    - Cada bloque try debe tener al menos un bloque catch correspondiente.
    - Las excepciones pueden ser lanzadas (o relanzadas) dentro de un bloque catch.
    - Se puede utilizar un bloque finally después de los bloques catch.
    - El objeto lanzado debe ser una instancia de la clase exception o de una subclase de exceptions. Intentar lanzar un objeto que no lo es resultara en un error fatal de PHP.
    - Cuando una excepción es lanzada, el código siguiente a la declaración no será ejecutado, y PHP intentará encontrar el primer bloque catch coincidente. Si una excepción no es capturada, se emitirá un error fatal de PHP con un mensaje

"Uncaught Exception…." (Excepción No Capturada), a menos que se haya definido un gestor con set_exception_handler().

ESTRUCTURAS DE CONTROL

Son el 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: se dividen en tres grupos.

- Secuencia:
  - Es el flujo de control más simple.
  - Indica la ejecución de una sentencia a continuación de otra.
  - El delimitador más general y más usado es el ";"
  - Hay lenguajes que no tienen delimitador y establecen que por cada línea vaya una instrucción. Se los llaman orientados a línea. Ej: Fortran, Basic, Ruby, Phyton.
  - Se pueden agrupar varias sentencias en una, llamada *Sentencia Compuesta*, llevan delimitadores como Begin y End. Ej: ADA, { y } en C, C++, Java, etc.
- Asignación:
  - Sentencia que produce cambios en los datos de la memoria.
  - Asigna al L-valor de un dato objeto el R-valor de una expresión.
  - Sintaxis en diferentes lenguajes.

A := B	Ej: Pascal, Ada, etc.
A = B	Ej: Fortran, C, Prolog, Python, Ruby, etc.
MOVE B TO A	COBOL
A← B	APL
(SETQ A B)	LISP

- Distinción entre sentencia y expresión:
  - En cualquier lenguaje convencional, existe diferencia entre sentencia de asignación y expresión.
  - En otros lenguajes tales como C definen la sentencia de asignación, como una expresión con efectos laterales.
  - Las sentencias de asignación devuelven valores.
  - Evalúa de derecha a izquierda en C
  - Ejemplo a=b=c=0;
  - If (i=30) printf("es verdadero")
  - La mayoría de los lenguajes de programación requieren que sobre el lado izquierdo de la asignación aparezca un l-valor. C permite cualquier expresión que denote un l-valor.
- Selección:
  - Esta estructura de control permite que el programador pueda expresar una elección entre un cierto número posible de sentencias alternativas.
  - Evolución:

    → If lógico de Fortran

    - If (condición lógica) sentencia

Si la condición es verdadera ejecuta la sentencia.
→ If then else de Algol

- If (condición lógica) then sentencia1
  Else sentencia2

Esto permite tomar dos caminos posibles

- Problemas:
  - Ambigüedad, no establecía por lenguaje como se asociaban los else con los if abiertos.
    - If then else de Pl/1, Pascal y C sin ambigüedad.
  - Establecen por lenguaje que cada else cierra con el último if abierto.
- Solución: sentencia de cierre del bloque condicional, por ejemplo end if, fi, etc.
- Desventajas: Ilegibilidad, programas con muchos if anidados pueden ser ilegibles. Utilizar Begin End.
  → If then else de Python, sin Ambigüedad y legible

```
if sexo == 'M':
    print 'La persona es Mujer'
else :
    print 'La persona es Varón'
```

```
if hora <= 6 and hora >=12:
    print 'Buenos días!!'
elif hora >12 and hora < 20 :
    print 'Buenas tardes!!'
else :
    print 'Buenas noches!!'
```

  - : es obligatorio al final del if, else y elif
  - La indentacion es obligatoria al colocar las sentencias correspondientes tanto al if, else y elif.
- Sentencia condicional – A if C else B de Phyton
  - Construcción equivalente al "?" del lenguaje C
  - Devuelve A si se cumple la condición C, sino devuelve B
  - Ejemplo:

```
>>> altura = 1.79
>>> estatura = "Alto" if altura > 1.65 else "Bajo"
>>> estatura
'Alto'
>>>
```

IMPORTANTE: Python para evaluar las condiciones utiliza la "Evaluación con circuito corto"

- Selección Múltiple:
  - Sentencia Select de Pl/1: Pl/1 incorpora la sentencia de selección entre dos o más opciones.
    Select
    When (a) sentencia1;
    When (b) sentencia2;
    ……
    Otherwise sentencia n;
    End;
    Reemplaza al if (A) then sentencia1
    Else if (B) then sentencia 2
    …….
  - Pascal:
    - Incorpora que los valores de la expresión sean ordinales y ramas con etiquetas. No importan el orden en que aparecen las ramas. Tiene la opción "else".
    - Es inseguro porque no establece que sucede cuando un valor no cae dentro de las alternativas puestas.

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

El else es opcional, que hace si lse ingresa un 5 y no hay un else?

- ADA:
  - Las expresiones pueden ser solamente de tipo entero o enumerativo.
  - En las selecciones se debe estipular todos los valores posibles que puede tomar la expresión.
  - Tiene la cláusula Others que se puede utilizar para representar a aquellos valores que no se especificación explícitamente.

    Si NO se coloca la rama para un posible valor o si NO aparece la opción Others en esos casos, no pasara la compilación.
  - Ejemplo 1:

    Case operador is
    When '+' => result:=a + b;
    When '-' => result:=a – b;
    When Others => result:=a * b;
    End case

    La cláusula others se debe colocar porque las etiquetas de las ramas NO abarcan todos los posibles valores de Operador.
  - Ejemplo 2:

    Case hoy is
    When MIE..VIE => Entrenar_duro; -- rango
    When MAR | SAB => Entrenar_poco; -- varias elecciones
    When DOM => Competir; -- una elección
    When Others => Descansar; -- debe ser única y la última alternativa
    End case;
- C, C++:
  - Constructor Switch
  - Cada rama es etiquetada por uno o más valores constantes
  - Cuando la opción coincide con una etiqueta del Switch se ejecutan las sentencias asociadas y se continúa con las sentencias de las otras entradas.
  - Existe la sentencia break, que provoca la salida.
  - Tiene una clausula default que sirve para los casos que el valor no coincida con ninguna de las opciones establecidas.

```
Switch Operador {
    case  '+' :
        result:= a + b; break;
    case  '-':
        result:= a - b;  break;
    default :
        result:= a * b;
    }
```

De be ponerse la sentencia **break** para saltar las siguientes ramas

- Ruby:

```ruby
raza = 'enano'
# Ahora utilizaremos el case
puts 'Utilizando case asignado a una varible :'
personaje = case raza
            when 'elfo' then 'Legolas'
            when 'enano' then 'Gimli'
            when 'mago' then 'Gandalf'
            when 'ents' then 'Barbol'
            when 'humano' then 'Aragorn'
            when 'orco' then 'Ufthak'
end
puts personaje
```

  - Si no entra en ninguna opción, sigue la ejecución y la variable no tendrá ningún valor.

- Iteración:
  - Este tipo de instrucciones se utilizan para representar aquellas acciones que se repiten un cierto número de veces.
  - Su evolución:
    - Sentencia Do de **Fortran**

      Do label var-de-control=valor1, valor2

      …….

      Label continue
  - La variable de control solo puede tomar valores enteros.
  - El Fortran original evaluaba si la variable de control había llegado al límite al final del bucle, o sea que siempre una vez lo ejecutaba.
    - Sentencia For de **Pascal, ADA, C, C++**
  - La variable de control puede tomar cualquier valor ordinal, no solo enteros.
  - Pascal estándar no permite que se toquen ni los valores del límite inferior y superior, ni el valor de la variable de control.
  - La variable de control puede ser de cualquier valor ordinal.
  - El valor de la variable fuera del bloque se asume indefinida.

Ada, no debe declararse el iterador

```
for i in 1..N loop
V(i) := 0;
end loop
```

Puede ser reverse

    - **C, C++** se compone de tres partes: una inicialización y dos expresiones.
  - La primer expresión (2do parámetro) es el testeo que se realiza ANTES de cada iteración. Si no se coloca el for queda en LOOP.
  - En el primer y último parámetro se pueden colocar sentencias separadas por comas.

```
for (p= 0, i=1 ; i<=n ; i++)
{
p+= a * b;
b = p * 8;
}
```

En C++ pueden haber declaraciones. Ej: int p=0,

    - **Phyton**, estructura que permite iterar sobre una secuencia. Las secuencias pueden ser: una lista, una tupla, etc.

```
lista = [ "el", "for", "recorre", "toda", "la", "lista"]
for variable in lista:
    print variable
```

Imprimirá:
el
for
recorre
toda
la
lista

Si se quiere que el for actúe como en los demás lenguajes, entonces hacer uso de la función "range()", ej. range(6). Hará que variable tome los valores de 0 a 5.

- **Phyton**, uso de la función “range()”
  - La función devuelve una lista de números enteros ej. range(5), devuelve [0,1,2,3,4].
  - Puede tener hasta 3 argumentos:
    - 2 argumentos: range(2,5), devuelve [2,3,4]
    - 3 argumentos: range(2,5,2), devuelve [2,4]

  Esa función range() da la posibilidad de simular la sentencia FOR de otros lenguajes
  For i in range (valor-inicial, valor-final + 1): acciones
  Ejemplo: for vuelta in range(1,10):
  Print(“vuelta “+str(vuelta))

- Iteración: While
  - Estructura que permite repetir un proceso mientras se cumpla una condición.
  - La condición se evalúa antes de que se entre al proceso.
    - En pascal:
      - While condición do sentencia;
    - En C, C++:
      - While (condición) sentencia;
    - En ADA:
      - While condición sentencia
        End loop;
    - En Phyton:
      - While condición:
        Sentencia1
        Sentencia2
        …
        Sentencia n
- Iteración: Until
  - Estructura que permite repetir un proceso mientras se cumpla una condición.
  - La condición se evalúa al final del proceso, por lo que por lo menos una vez el proceso se realiza.
    - En Pascal:
      - Repeat
        Sentencia
        Until condición;
    - En C, C++:
      - Do sentencia;
        While (condición);
- Iteración: Loop
  - ADA tiene una estructura iterativa

    Loop
    …
    End loop;
  - De este bucle se sale normalmente, mediante una sentencia “exit when” o con una alternativa que contenga una clausula “exit”.

    Loop
    …
    Exit when condición;
    …
    End loop;

PARADIGMAS

Un **paradigma de programación** es un estilo de desarrollo de programas, un modelo para resolver problemas computacionales. Los lenguajes de programación, necesariamente, se encuadran en uno o varios paradigmas a la vez, a partir del tipo de órdenes que permiten implementar, tiene una relación directa con su sintaxis.

- Principales paradigmas:
  - **Imperativo:** sentencias + secuencias de comandos.
  - **Declarativo:** los programas describen los resultados esperados sin listar explícitamente los pasos a llevar a cabo para alcanzarlos.
  - **Lógico:** aserciones lógicas: hechos + reglas, es declarativo.
  - **Funcional:** los programas se componen de funciones.
  - **Orientado a objetos:** métodos + mensajes.
- Otra forma de clasificación más reciente:
  - **Dirigido por eventos:** el flujo del programa está determinado por sucesos externos (por ejemplo, una acción del usuario)
  - **Orientado a aspectos:** apunta a dividir el programa en módulos independientes, cada uno con un comportamiento y responsabilidad bien definido.
- Paradigma aplicativo o funcional:
  - Basado en el uso de funciones. Muy popular en la resolución de problemas de inteligencia artificial, matemática, lógica, pensamiento paralelo.
  - Ventajas:
    - Vista uniforme de programa y función.
    - Tratamiento de funciones como datos.
    - Liberación de efectos colaterales.
    - Manejo automático de memoria.
  - Desventajas:
    - Ineficiencia de ejecución.
- Paradigma funcional:
  - Características:
    - Define un conjunto de datos.
    - Provee un conjunto de funciones primitivas.
    - Provee un conjunto de formas funcionales.
    - Requiere de un operador de aplicación.
    - Semántica basada en valores.
    - Transparencia referencial.
    - Regla de mapeo basada en combinación o composición.
    - Las funciones de primer orden.
- Programación funcional:
  - Funciones:
    - El VALOR más importante en la programación es el de una FUNCION.
    - Matemáticamente una funciones es una correspondencia: f: A → B
    - A cada elemento de A le corresponde un único elemento en B.
    - f(x) denota el resultado de la aplicación de f a x
    - Las funciones son tratadas como valores, puede ser pasadas como parámetros, retornar resultados, etc.
  - Definiendo funciones:
    - Se debe distinguir entre el VALOR y la DEFINICION de una función.
    - Existen muchas maneras de DEFINIR una misma función, pero siempre dará el mismo valor, ejemplo: DOBLE X = X + X DOBLE' X = 2 * X
      Denotan la misma función pero son dos formas distintas de definirlas.
  - Tipo de una función:
    - Puede estar definida explícitamente dentro del SCRIPT, por ejemplo:
      Cuadrado::num → num
      Cuadrado x = x + x
    - O puede **deducirse/inferirse** el tipo de una función.
  - Expresiones y valores:
    - La expresión es la noción central de la programación funcional.
    - Característica más importante: "Una expresión es su VALOR"

- El valor de una expresión depende únicamente de los valores de las sub expresiones que la componen.
- Las expresiones también pueden contener VARIABLES, (valores desconocidos).
- La noción de Variable es la de "variable matemática", no la de celda de memoria. "Diferentes ocurrencias del mismo nombre hacen referencia al mismo valor desconocido"
- Las expresiones cumplen con la propiedad de "transparencia referencial": dos expresiones sintácticamente iguales darán el mismo valor, porque no existen efectos laterales.
- Ejemplo de expresiones para evaluar:
  - Expresión Valor
    47 47
    (*4 7) 28 Se está utilizando una función primitiva (el * y +)
    (+49 5) 54}

o Definiendo funciones:
- Cuadrado x = x * x
  Min x y = x, if x < y
  y, if x > y
  cube (x) = x * x * x

o Un script es una lista de definiciones y:
- Pueden someterse a evaluación: Ejemplos: ?cuadrado (3 + 4) ?min 3 4
  49 3
- Pueden combinarse, ejemplo: ?min(cuadrado (1 + 1) 3)
- Pueden modificarse, ejemplo: Al script anterior le agrego nuevas definiciones:
  Lado = 12
  Área = cuadrado lado,
- Alunas expresiones pueden NO llegar a reducirse del todo, ejemplo: 1/0
- A esas expresiones se las denominan **canónicas,** pero se les asigna un **valor definido** y corresponde al símbolo bottom(^)
- Por lo tanto toda **expresión** siempre nota un **valor**

o Evaluación de las expresiones:
- La forma de evaluar es a través de un mecanismo de REDUCCION o SIMPLIFICACION.
- Ejemplo:
  - Cuadrado (3 + 4)
    - Cuadrado 7 (+)
    - 7 * 7 (cuadrado)
    - 49 (*)
  - Otra forma seria: cuadrado (3 + 4)
    - ➔ (3 + 4) * (3 + 4) (cuadrado)
    - ➔ 7 * (3 + 4) (+)
    - ➔ 7 * 7 (+)
    - ➔ 49 (*)

  "no importa la forma de evaluarla, siempre el resultado final será el mismo"

o Existen dos formas de reducción:
- Orden aplicativo: aunque no lo necesite SIEMPRE evalúa los argumentos.
- Orden normal (lazy evaluation): no calcula más de lo necesario. La expresión NO es evaluada hasta que su valor se necesite. Una expresión compartida NO es evaluada más de una vez.

o Tipos: básicos y derivados.
- Básicos: son los primitivos, ejemplo:
  - NUM (INT y FLOAT) (números)
  - BOOL (valores de verdad)
  - CHAR (caracteres)
- Derivados: se construyen de otros tipos, ejemplo:
  - (num,char) tipo de pares de valores.

- (num→char) tipo de una función.
  TODA FUNCION TIENE ASOCIADO UN TIPO

- Expresiones de tipo polimórficas:
  - En algunas funciones no es tan fácil deducir su tipo.
    - Ejemplo: id x = x
      Esta función es la función Identidad
      Su tipo puede ser de char → char, de num → num, etc.
      Por lo tanto su tipo será B → B
      Se utilizan letras griegas para tipos polimórficos.
    - Otro ejemplo: letra x = "A"
      Su tipo será B → char
- Currificacion:
  - Mecanismo que reemplaza argumentos estructurados por argumentos más simples.
  - Ejemplo: sean dos definiciones de la función "suma"
    1- Suma (x,y) = x + y
    2- Suma' x y = x + y
    Existen entre estas dos definiciones una diferencia sutil: "diferencia de tipos de función"
    El tipo de suma es: (num,num) → num
    El tipo de suma' es: num → (num→num) // por cada valor de x devuelve una función
    Aplicando la función:
    Suma (1,2) → 3
    Suma' 1 2 → Suma'1 aplicado al valor 2 // para todos los valores devuelve el siguiente
- Calculo Lambda:
  - Es un modelo de computación para definir funciones.
  - Se utiliza para entender los elementos de la programación funcional y la semántica subyacente, independientemente de los detalles sintácticos de un lenguaje de programación en particular.
  - Las expresiones del Lambda calcula pueden ser de 3 clases:
    - Un simple identificador o una constante. Ej: x, 3
    - Una definición de una función. Ej: λ x.x+1
    - Una aplicación de una función. La forma es (e1 e2), donde se lee e1 se aplica a e2. Ej: en la función cube (x) = x * x * x
      Λ x.x * x * x (λ x.x * x * x) 2) //evaluamos la función con 2 y resulta en 8

- Programación lógica:
  - Es un paradigma en el cual los programas son una seria de aserciones lógicas.
  - El conocimiento se representa a través de **reglas** y **hechos**.
  - Los objetos son representados por **términos**, los cuales contienen constantes y variables.
  - PROLOG es el lenguaje lógico más utilizado.
  - Elementos de la programación lógica:
    - La sintaxis básica es el **"termino".**
    - Variables:
      - Se refieren a elementos indeterminados que pueden sustituirse por cualquier otro.
        "humano (X)", la X puede ser sustituida por constantes como: juan, pepe, etc
      - Los nombres de las variables comienzan con mayúsculas y pueden incluir números.
    - Constantes:
      - A diferencia de las variables son elementos determinados.
        "humano (juan)"
      - Las constantes son string de letras en minúscula (representan objetos atómicos) o string de dígitos (representan números)
    - Termino compuesto:
      - Consisten en un "functor" seguido de un número fijo de argumentos encerrados entre paréntesis, los cuales son a su vez términos.

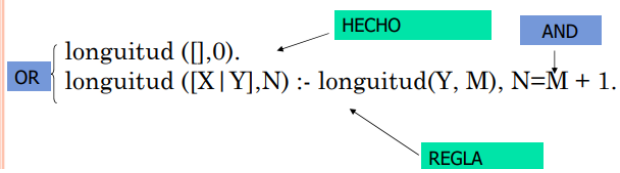
- Se denomina “aridad” al número de argumentos.
- Se denomina “estructura” (ground term) a un término compuesto cuyos argumentos no son variables.

▪ Ejemplos:
- Padre → constante
- Longitud → variable
- tamaño(4,5) → estructura

▪ Listas:
- La constante [] representa una lista vacía.
- El functor “.” Construye una lista de un elemento y una lista. Ejemplo: .(alpha,[]), representa una lista que contiene un único elemento que es alpha.
- Otra manera de representar la lista es usando [] en lugar de .(). Ejemplo anterior la lista quedaría: [alpha,[]]
- Y también se representa utilizando el símbolo | . Ejemplo: [alpha | []]
  La notación general para denotar lista es : [X|Y]
  X es el elemento cabeza de la lista e
  Y es una lista, que representa la cola de la lista que se está modelando

o Cláusulas de Horn:
▪ Un programa escrito en un lenguaje lógico es una secuencia de “clausulas”.
▪ Las clausulas pueden ser: un “Hecho” o una “Regla”.
- Hecho:
  - Expresan relaciones entre objetos
  - Expresan verdades
  - Son expresiones del tipo p(t1,t2,…,tn)
  - Ejemplos:
    - tiene(coche,ruedas) → representa el hecho que un coche tiene ruedas
    - longitud([],0) → representa el hecho que una lista vacía tiene longitud cero
    - virus(ithaqua) → representa el hecho que ithaqua es un virus
- Regla:
  - Cláusula de Horn
  - Tiene la forma: conclusión :- condición
  - Donde:
    - :- indica “Si”
    - Conclusión es un simple predicado y
    - Condición es una conjunción de predicados, separados por comas. Representan un AND lógico.
  - Una regla PROLOG conclusión:-condiciones.
  - En un lenguaje procedural una regla la podríamos representar como: if condición else conclusión;
  - Ejemplo: virus (X) :- programa(X),propaga(X)

- <u>Programas y Queries</u>
  - Ejemplo de programa:



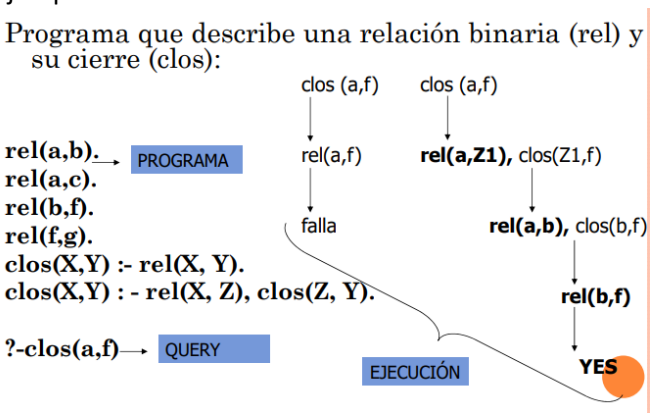
  - <u>Programa</u>: conjunto de cláusulas.
    ▪ ?-longitud([rojo| [verde | [azul | []]]], X).
  - <u>Query</u>: representa lo que deseamos que sea contestado.

Programa:

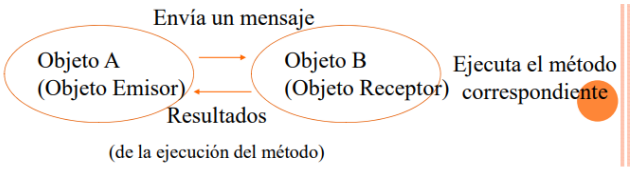
```
longuitud ([],0).
longuitud ([X|Y],N) :- longuitud(Y, M), N=M + 1.

?-longuitud([rojo| [verde | [azul | [] ] ] ] ,X).
```

longuitud([verde | [azul | []]],M) y X=M+1
longuitud([azul | []] ,Z) y M=Z+1
longuitud([],T) Z=T+1 T=0 => Z=1
M=2
X=3

- Ejecución de programas
  - Un programa es un conjunto de reglas y hechos que proveen una especificación declarativa de que es lo que se conoce y la pregunta es el objetivo que queremos alcanzar.
  - La ejecución de dicho programa será el intento de obtener una respuesta.
  - Desde un punto de vista lógico la respuesta a esa pregunta es "YES", si la pregunta puede ser derivada aplicando "deducciones" del conjunto de reglas y hechos dados.
  - Ejemplo:



- Programación Orientada a Objetos
  - "un programa escrito con un lenguaje OO es un conjunto de objetos que **interactúan** mandándose **mensajes**"
  - Los elementos que intervienen en la programación OO son:
    - Objetos:
      - Son entidades que poseen estado interno y comportamiento.
      - Es el equivalente a un dato abstracto.
    - Mensajes:
      - Es una petición de un objeto a otro para que este se comporte de una determina manera, ejecutando uno de sus métodos.
      - TODO el procesamiento en este modelo es activado por mensajes entre objetos.
    - Métodos:
      - Es un programa que está asociado a un objeto determinado y cuya ejecución solo puede desencadenarse a través de un mensaje recibido por este o por sus descendientes.



    - Clases:
      - Es un tipo definido por el usuario que determina las estructuras de datos y las operaciones asociadas con ese tipo.
      - Cada objeto pertenece a una clase y recibe de ella su funcionalidad.
      - Primer nivel de abstracción de datos: definimos estructura, comportamiento y tenemos ocultamiento.

        - La información contenida en el objeto solo puede ser accedida por la ejecución de los métodos correspondientes.
    - Instancia de clase:
        - Cada vez que se construye un objeto se está creando una INSTANCIA de esa clase.
        - Una instancia es un objeto individualizado por los valores que tomen sus atributos.
- Otro aspecto de las abstracciones de dato
  GENERALIZACION/ESPECIFICACION → HERENCIA
  El segundo nivel de abstracción consiste en agrupar las clases en jerarquías de clases (definiendo SUB y SUPER clases), de forma tal que una clase A here todas las propiedades de su superclase B (suponiendo que tiene una).
- Otros conceptos adicionales:
    - Polimorfismo:
        - Es la capacidad que tienen los objetos de distintas clases de responder a mensajes con el mismo nombre.
        - Ejemplo: 3 + 5 se aplica suma entre números
          "buenos" + "días" se concatenan strings
    - Binding dinámico:
        - Es la vinculación en el proceso de ejecución de los objetos con los mensajes.
- C++ (lenguaje hibrido): algunas características.
    - Lenguaje extendido del lenguaje C
    - Incorporo características de POO
    - Lenguaje compilativo el ambiente de programación es de los lenguajes tradicionales.
    - Los objetos en C:
        - Se agrupan en tipos denominados clases
        - Contienen datos internos que definen su estado interno
        - Soportan ocultamiento de datos
        - Los métodos son los que definen su comportamiento
        - Pueden heredar propiedades de otros objetos
        - Pueden comunicarse con otros objetos enviándose mensaje

LENGUAJES DE SCRIPTING

- Los lenguajes de programación tradicionales están destinados principalmente para la construcción de aplicaciones auto-contenidas:
    - Programas que aceptan una suerte entrada, la procesan de una manera bien entendida y finalmente generan una salida apropiada.
    - Sin embargo, muchos de los usos actuales en diferentes entornos, requieren la coordinación de múltiples programas.
- Lenguajes de programación tradicionales VS LBS
    - Los lenguajes convencionales tienden a **mejorar eficiencia**, mantenibilidad, portabilidad y detección estática de errores. Los tipos se construyen alrededor de conceptos a nivel hardware como enteros de tamaño fijo, punto flotante, caracteres y arreglos.
    - Los lenguajes script tienden a **mejorar flexibilidad**, desarrollo rápido y chequeo dinámico. Su sistema de tipos se construye sobre conceptos de más alto nivel como tablas, patrones, listas y archivos.
- LBS
    - Los lenguajes script de propósito general (Perl, Python) suelen conocerse como glue-lenguages.
    - Se diseñaron para "pegar" programas existentes a fin de construir un sistema más grande.
    - Se utilizan como lenguajes de extensión, ya que permiten al usuario adaptar o extender las funcionalidades de las herramientas script.
    - Son interpretados
    - En general, débilmente tipados (tipificados), por lo tanto, muy flexibles, aunque suelen ser menos eficientes en la ejecución.
- ¿Qué es un lenguaje Script?

    - Es difícil definirlos con precisión, aunque hay varias características que tienden a tener en común.
    - Estos lenguajes tienen dos tipos de ancestros:
        - Intérpretes de líneas de comando o "shells"
        - Herramientas para procesamiento de texto y generación de reportes.
    - "Los lenguajes script asumen la existencia de componentes útiles en otros lenguajes. Su intención no es escribir aplicaciones desde el comienzo, sino por combinación de componentes" (John Ousterhout – creador de TCL)
- Aspectos principales
    - En los LBS las declaraciones son escasas o nulas y proveen reglas simples que gobiernan el alcance de los identificadores.
    - Por ejemplo, en Perl, cada identificador es global por omisión (hay opciones para limitar el alcance)
    - En otros lenguajes (e.g., PHP, Tcl), cada cosa es local por omisión (un objeto global debe ser explícitamente importado).
    - Python adopta la regla: "a una variable que se le asigna un valor es local al bloque donde aparece dicha asignación" (se puede cambiar esta regla con una sintaxis especial)
    - Dado la falta de declaraciones, muchos LBS incorporan tipificación dinámica.
    - En algunos lenguajes el tipo de la variable es chequeada inmediatamente antes de su uso: e.g., PHP, Python, Ruby, y Scheme.
    - En otro, el tipo de una variable (por ende su valor) será interpretado de manera diferente según el contexto: e.g., Rexx, Perl, y Tcl.
- Facilidades avanzadas para:
    - Procesamiento de texto y generación de reportes.
    - Manipulación de Entrada/Salida de programas externos
    - Pattern matching, búsqueda y manipulación
    - La mayoría de los comandos están basados en Expresiones Regulares Extendidas (dir *.*)
- Tipos de datos de alto nivel
    - Los LBS son ricos en:
        - Conjuntos
        - Diccionarios
        - Listas
        - Tuplas, etc.
    - Por ejemplo:
        - En muchos LBS es común poder indexar arreglos a través de cadenas de caracteres, lo que implica una implementación de tablas de hash y manejo de almacenamiento usando "garbage collection".
- Dominios de aplicación (general)
    - Principales ejemplos:
        - Lenguaje de comandos (Shell)
        - Procesamiento de texto y generación de reportes
        - Matemática y estadística
        - Lenguajes de "pegado" (GLUE) y de propósito general
        - Extensión de lenguajes
        - www como ejemplo especial
            - CGI (Common Gateway Interface)
            - Scripts embebidos en servidores
            - Scripts embebidos en clientes
            - Applets de java
            - Otros: XML.

ETICA

- 1) Disciplina filosófica que estudia el bien y el mal y sus relaciones con la moral y el comportamiento humano.
- 2) Conjunto de costumbres y normas que dirigen o valoran el comportamiento humano en una comunidad.

- La **ética profesional** aparece recogida en los códigos deontológicos que regulan una actividad profesional. La **deontología** forma parte de la que se conoce como **ética normativa** y presenta una serie de principios y reglas de cumplimiento obligatorio.
- Ética en informática
    - La tecnología informática plantea nuevas situaciones y nuevos problemas y gran parte de estas, son de una naturaleza ética.
    - "En las actividades profesionales relacionadas con las tecnologías informáticas se quiere pasar de la simple aplicación de criterios éticos generales a la elaboración de una ética propia de la profesión. Los códigos éticos de asociaciones profesionales y de empresas de informática van en esa direccion".
    - La ética informática tiene varios objetivos, algunos de ellos son:
        - Descubrir y articular dilemas éticos claves en informática.
        - Determinar en qué medida son agravados, transformados o creados por la tecnología informática.
        - Proponer un marco conceptual adecuado para entender los dilemas éticos que origina la informática y, además, establecer una guía cuando no existe reglamentación de dar uso a internet.
- Dimensiones sociales de la informática
    - Desarrollo de los medios de comunicación en la sociedad.
    - Globalización de la economía (pérdidas de puestos de trabajo, desigualdad…)
    - Investigación, desarrollo y producción de tecnología.
- Conclusiones
    - Toda la actividad del hombre debe ser regida por un código de ética y la información no es la excepción.