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.

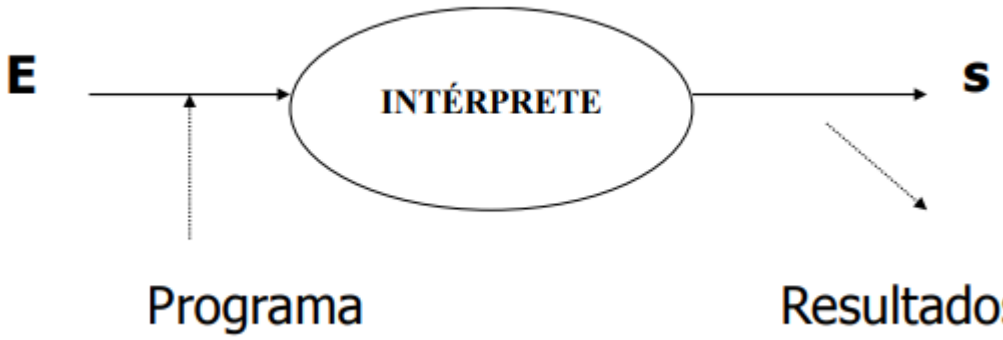
PROCESAMIENTO DE UN LENGUAJE

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”

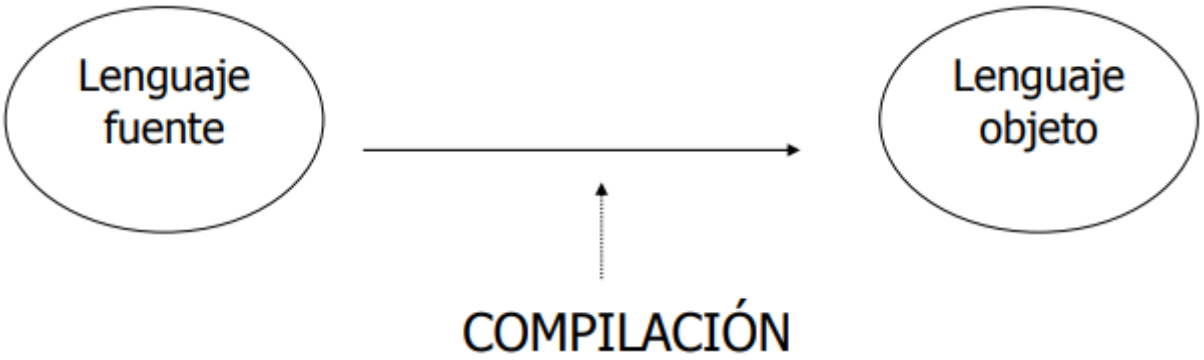
INTERPRETACION Y COMPILACION

- 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:

    …

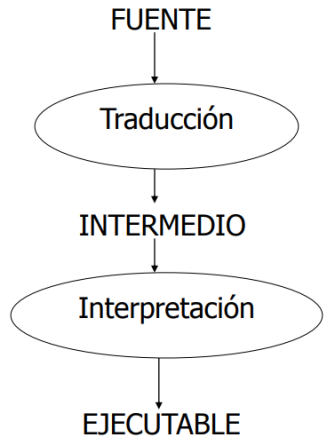
    r = max(s,5);

    ….
  - 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.



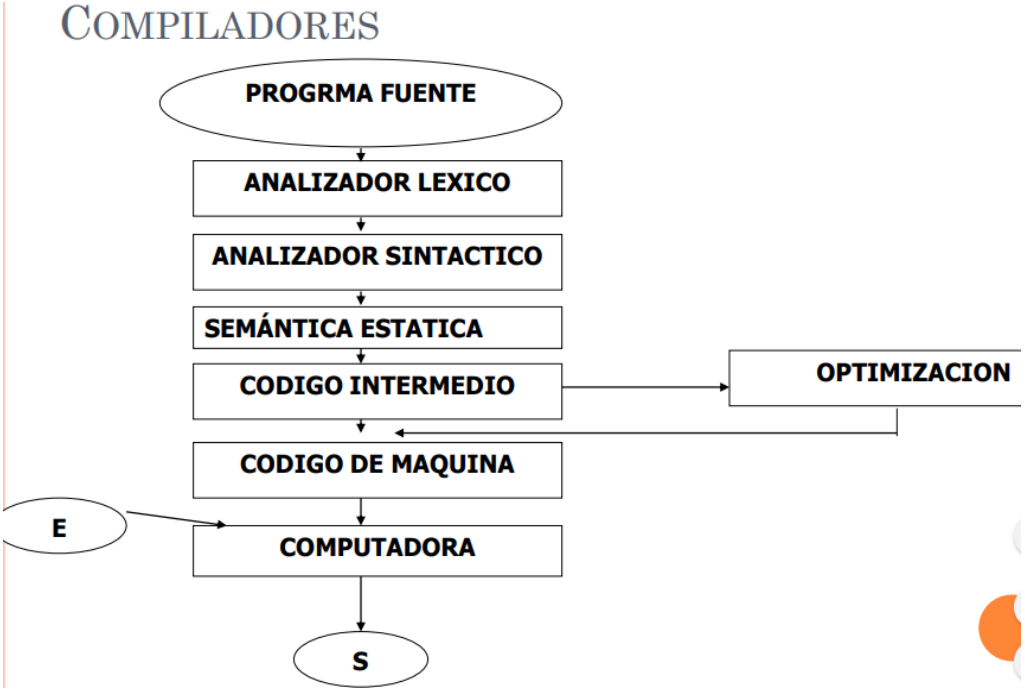
- Compiladores
  - Al compilar los programas la ejecución de los mismos es más rápida. Ej. de programas que se compilan: C, Ada, Pascal, etc.
  - Los compiladores pueden ejecutarse en un solo paso o en dos pasos.
  - En ambos casos cumplen con varias etapas, las principales son:
    - Análisis
      - 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

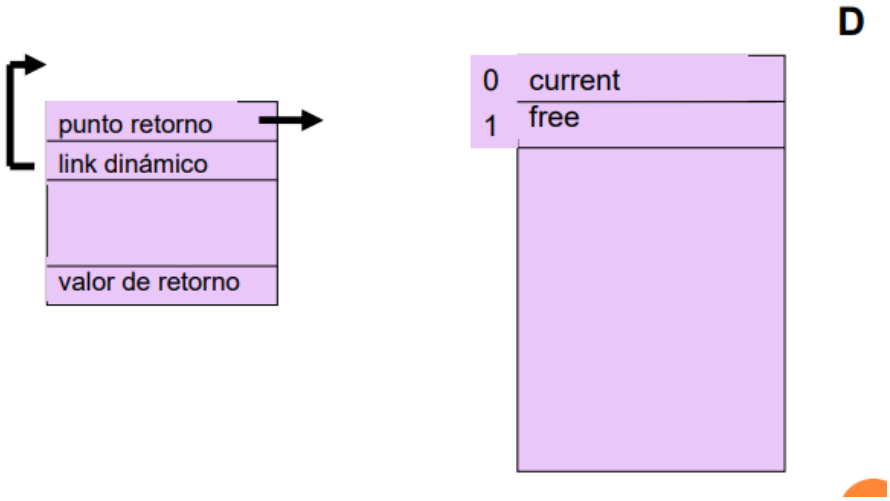
- C2'
    - El compilador no puede ligar variables locales a direcciones absolutas
    - Tampoco variables globales
    - Linkeditor:
        - Encargado de combinar los módulos
        - Ligar la información faltante
    - C2 y C2' no difieren semánticamente.

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)