Resumen Primera EMT CPLP

Para esta EMT se evalúan las clases 1, 2 y 3 de teoría de la materia.

Clase 1

Lenguajes de Programación

- Son el corazón de la Ciencia de la Informática, funcionan como herramientas para comunicarnos con las máquinas a través de programas y también con las personas.
- El valor de los mismos se tiene que juzgar según cómo afectan a la producción de Software y a la facilidad que tienen para integrarse con otras herramientas.

Criterios para evaluar los Lenguajes de Programación

Simplicidad y Legibilidad

- Los lenguajes deben poder ser simples o fáciles de escribir, si lo son, permiten generar programas fáciles de leer, es decir, más legibles. Otro aspecto que importa es que sean fáciles a la hora de aprenderlos o enseñarlos.
- Factores que atentan contra este criterio:
    - **Poseer muchos componentes elementales** (componentes del lenguaje, como el manejo de excepciones por ejemplo), si un lenguaje es muy extenso puede generar otro problema que es que las personas **conozcan subconjuntos de componentes** y no lleguen a conocer la totalidad de las mismas.

    - **Que el mismo concepto semántico tenga distintas sintaxis**, por ejemplo, una variable se puede incrementar de varias formas dependiendo el lenguaje “a = a + 1” o “a += 1” o “a++”, etc**.**
    - **Que distintos conceptos semánticos tengan la misma sintaxis.**
    - **Que haya un abuso de operadores sobrecargados**, es decir, para un mismo operador por ejemplo el “+” se le de uso para suma de enteros, reales, concatenación de strings, etc.

Claridad en los Bindings

- Una ligadura o binding se genera entre los elementos de los lenguajes de programación y sus atributos o propiedades, por ejemplo, si yo declaro una variable de tipo entero, el elemento que es la variable va a tener que ligarse con el atributo que es el tipo entero. Esta ligadura debe ser clara, es decir, el lenguaje en su definición tiene que proveernos información de cuándo se va a producir la ligadura, esta puede pasar en diferentes momentos:
    - **Binding Estático:** La ligadura se hace en Compilación.
    - **Binding Dinámico:** La ligadura se hace en Ejecución.
    - **En la definición del lenguaje.**
    - **En la implementación del lenguaje.**
    - **En el cargado del programa.**
    - **En la escritura del programa.**

Confiabilidad

- La Confiabilidad de los lenguajes está relacionada con la seguridad que ofrece frente a 2 aspectos:
    - **Chequeo de Tipos:** Los errores relacionados con los tipos deben ser encontrados cuanto antes para que sea menos costoso.
    - **Manejo de Excepciones:** Los lenguajes que ofrezcan alguna estructura para el manejo de excepciones que permita la detección de las mismas en tiempo de ejecución en pos de tomar medidas correctivas y continuar, serán más confiables.

Soporte

- Los lenguajes deberían ser accesibles para cualquier persona.
- Se deberían de poder implementar en diferentes plataformas.
- Deberían de poseer documentación y medios para familiarizarse con el lenguaje.

Abstracción

- Los lenguajes deben poder ofrecer abstracciones a nivel de procesos y datos mediante estructuras u operaciones que permitan ignorar muchos de los detalles.

Ortogonalidad

- Un lenguaje es ortogonal si ofrece un conjunto de elementos que puede ser combinado para crear estructuras de control y datos donde cada combinación es válida dentro del lenguaje y tiene sentido.

Eficiencia

- Se piensa la eficiencia de los lenguajes a partir de 3 aspectos:
    - **Tiempo y Espacio.**
    - **Esfuerzo Humano.**
    - **Si es Optimizable.**

Sintaxis

- Es el conjunto de reglas que definen como componer letras, dígitos y otros caracteres en pos de formar palabras y sentencias válidas dentro del lenguaje que nos permitan generar programas válidos.

Características de la Sintaxis

- Debe ayudar al programador a escribir programas correctos sintácticamente.
- Establece reglas que sirven para que el programador se comunique con el procesador.
- La Sintaxis debe:
    - **Ser Legible**.
    - **Ser Fácil de verificar y traducir** para los intérpretes y compiladores.
    - **No tener ambigüedades.**

Elementos de la Sintaxis

Alfabeto o conjunto de caracteres

- Conjunto de caracteres con los que trabaja el lenguaje. El que se utiliza actualmente es Unicode.
- Es importante el orden de los caracteres a la hora de realizar comparaciones.

Identificadores

- Hace referencia a los nombres de las rutinas/funciones/procesos, variables, etc.
- La elección más utilizada es una cadena de letras y dígitos que deben de comenzar con una letra.
- Si se les restringe la longitud, pierden legibilidad.

Operadores

- Normalmente hay consenso establecido para los operadores que se utilizan para las sumas, restas, etc. Pero pueden haber operadores como el mayor, menor, distinto, etc. que puede variar según el lenguaje.

Comentarios

- Símbolos especiales que permitan el uso de comentarios en el código aporta a la creación de programas más legibles.

Palabra Clave y Palabra Reservada

- **Palabra Clave:** Son palabras que tienen un significado específico dentro del lenguaje de programación, como por ejemplo Array, do, else, if, etc.
- **Palabra Reservada:** Son **palabras claves** que no pueden ser utilizadas por el programador para otro propósito que no sea el que haya sido especificado el lenguaje.
- **Ventajas del uso de estas palabras:**
    - Permiten al compilador y al programador expresarse claramente
    - Mejoran la legibilidad y agilizan la traducción.

Estructura Sintáctica

Vocabulario o words

- Conjunto de caracteres y palabras necesarias para construir expresiones, sentencias y programas.

Expresiones

- Funciones que a partir de un conjunto de datos, devuelve un resultado.
- Son consideradas bloques sintácticos.

Sentencias

- Componente Sintáctico más importante.
- Influyen fuertemente en la facilidad de escritura y la legibilidad.

Reglas Léxicas y Sintácticas

- **Reglas Léxicas:** Conjunto de reglas que hay que seguir para formar las **words**, a partir de los caracteres que nos ofrece el alfabeto. Por ejemplo, reglas para poder crear un identificador, si el lenguaje es case sensitive (diferencia entre mayúsculas y minúsculas), etc.
- **Reglas Sintácticas:** Conjunto de reglas que hay que seguir para formar **expresiones y sentencias** a partir del uso de las **words.**

Tipos de Sintaxis

- **Abstracta:** Se refiere a la estructura.
- **Concreta:** Se refiere a la parte léxica.
- **Pragmática:** Se refiere al uso práctico.

Cómo definir la sintaxis

- Usando **Lenguaje Natural.**
- Usando la **Gramática libre de contexto**, definida por Backus y Naur: **BNF.**
- Usando **Diagramas Sintácticos** que son equivalentes a **BNF.**

BNF (Backus Naur Form)

- Notación formal que se usa para describir la sintaxis.
- Es un metalenguaje que utiliza metasímbolos (<, >, ::=, |).
- Define las reglas mediante **producciones.**

Gramática

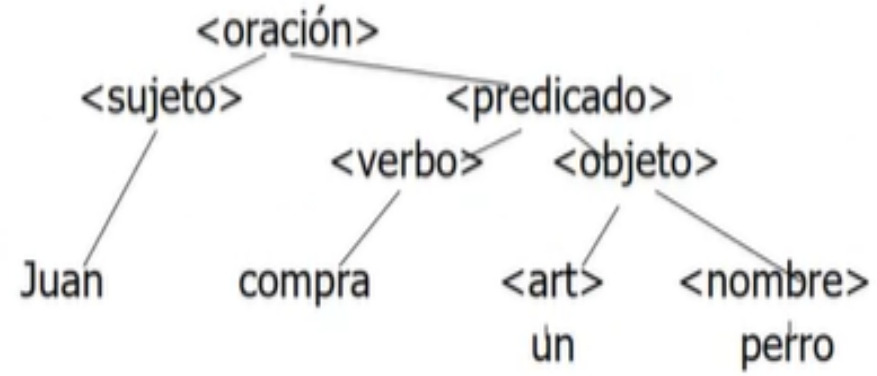
- Conjunto de reglas finitas que define un conjunto infinito de posibles sentencias válidas para el lenguaje.
- Está formada por una 4-tupla **G = (N, T, S, P)**
    - **N:** Conjunto de símbolos no terminales.
    - **T:** Conjunto de símbolos terminales.
    - **S:** Símbolo distinguido de la gramática perteneciente a **N.**
    - **P:** Conjunto de producciones.

Árboles Sintácticos

- Hace falta tener un **método de análisis** que nos permita determinar si un string es válido o no para nuestro lenguaje. Este proceso se denomina **Parse**, el cual para realizar este análisis construye para cada sentencia un **Árbol Sintáctico o de Derivación.**

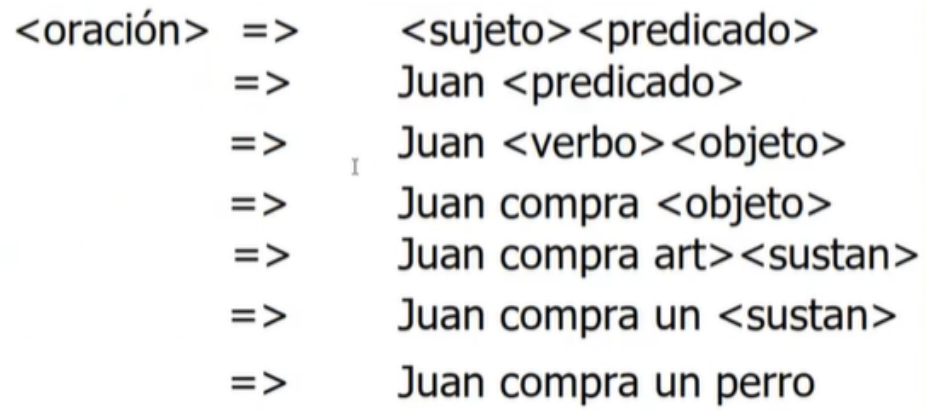
Maneras de Construirlo

- **Método bottom-up:** De izquierda a derecha o viceversa.
- **Método top-down:** De izquierda a derecha o viceversa.



Ejemplo de Árbol Sintáctico Top-down de izquierda a derecha

Árbol de Derivación



Ejemplo de Árbol de Derivación top-down de izquierda a derecha

Producciones Recursivas

- Hacen que el conjunto de sentencias que se describe sea infinito.
- Cualquier gramática que tenga una de estas, describe un **lenguaje infinito.**
- Ocurre cuando en una producción de un símbolo no terminal X aparece X en la definición de la misma.

Reglas Recursivas

- **Regla recursiva por la izquierda:** La asociatividad debe ocurrir por la izquierda, es decir, el símbolo no terminal de la parte izquierda tiene que aparecer al comienzo de la parte derecha. Por ejemplo: **<natural> ::= <natural><dígito> | <dígito>.**
- **Regla recursiva por la derecha:** La asociatividad debe ocurrir por la derecha, es decir, el símbolo no terminal de la parte izquierda tiene que aparecer al final de la parte derecha. Por ejemplo: **<natural> ::= <dígito> | <dígito><natural>.**

Gramáticas Ambiguas

- Una gramática es ambigua si una sentencia puede derivarse de más de una forma, es decir, puedo armar la misma sentencia de más de una forma distinta.

Subgramáticas

- Son normalmente utilizadas por compiladores ya que trabajan a bajo nivel.
- Son de utilidad ya que por ejemplo, para definir una gramática de expresiones se podrían utilizar las gramáticas de números y de identificadores.

Gramáticas libres y sensibles al contexto

- **Gramáticas Libres al contexto:** Son aquellas en donde no se realiza un análisis de contexto o semántico, es decir, pueden haber sentencias válidas sintácticamente pero que no tengan sentido semánticamente hablando. Las de **BNF y EBNF** entran en este grupo.
- **Gramáticas Sensibles al contexto:** Son las que analizan el contexto o semántica.

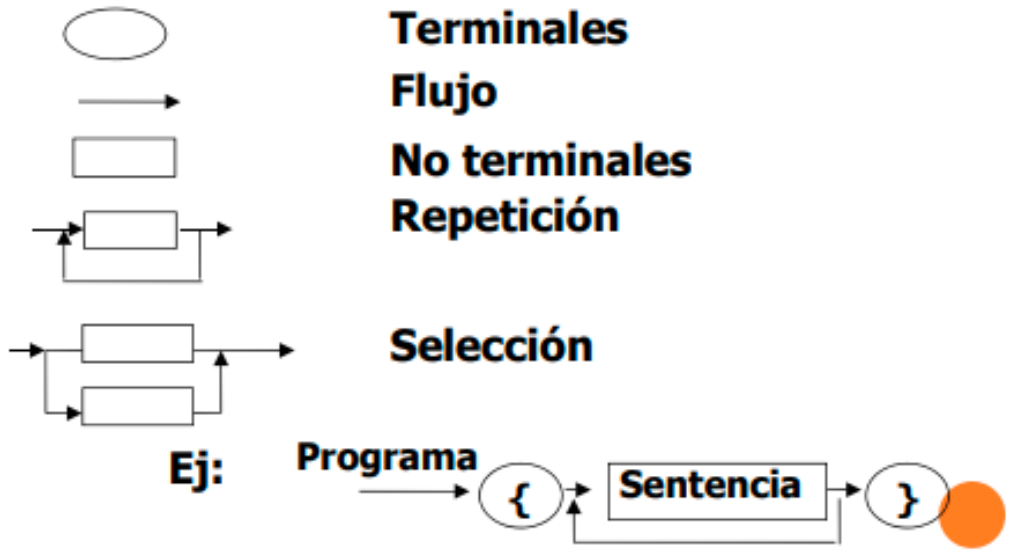
EBNF (Extended Backus Naur Form)

- Es la versión extendida de la BNF.
- Es más fácil de entender.
- Incorpora los siguientes símbolos:
    - **[ ]:** Elemento optativo.
    - **(|):** Selección de una alternativa.
    - **{}*:** Repetición de 0 o más veces.
    - **{}+:** Repetición de 1 o más veces.

Diagramas Sintácticos (CONWAY)

- Es un grafo sintáctico.
- Cada diagrama posee una entrada, una salida, y el camino es el análisis.
- Cada diagrama representa una producción.
- Para que una sentencia sea válida, debe haber un camino desde la entrada hasta la salida que la describa.
- Es una forma visual que se entiende mejor que **BNF o EBNF.**

Elementos del Diagrama



Clase 2

Semántica

- Describe el significado de los **símbolos**, **palabras** y **frases** de un **lenguaje** natural o informático que es **sintácticamente válido.**
- Sirve para darle significado a una construcción del lenguaje.

Semántica Estática

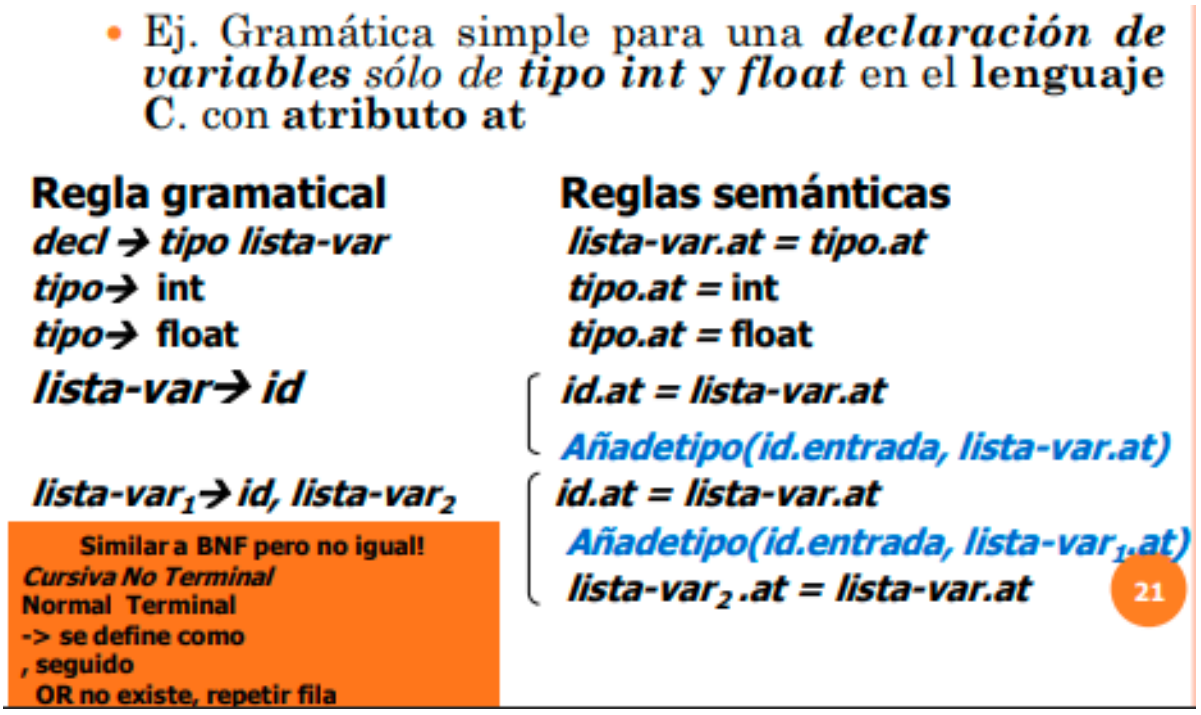
- No está relacionada con el significado de la ejecución del programa, sino con las formas válidas, es decir, con la sintaxis.
- Este proceso se realiza posterior al análisis sintáctico y previo al análisis de semántica dinámica.
- El análisis se hace en compilación, previo a la ejecución del programa, en pos de que el mismo no se pare una vez se esté ejecutando.
- No utiliza **BNF o EBNF** sino que usa **Gramática de Atributos.**

Gramática de Atributos

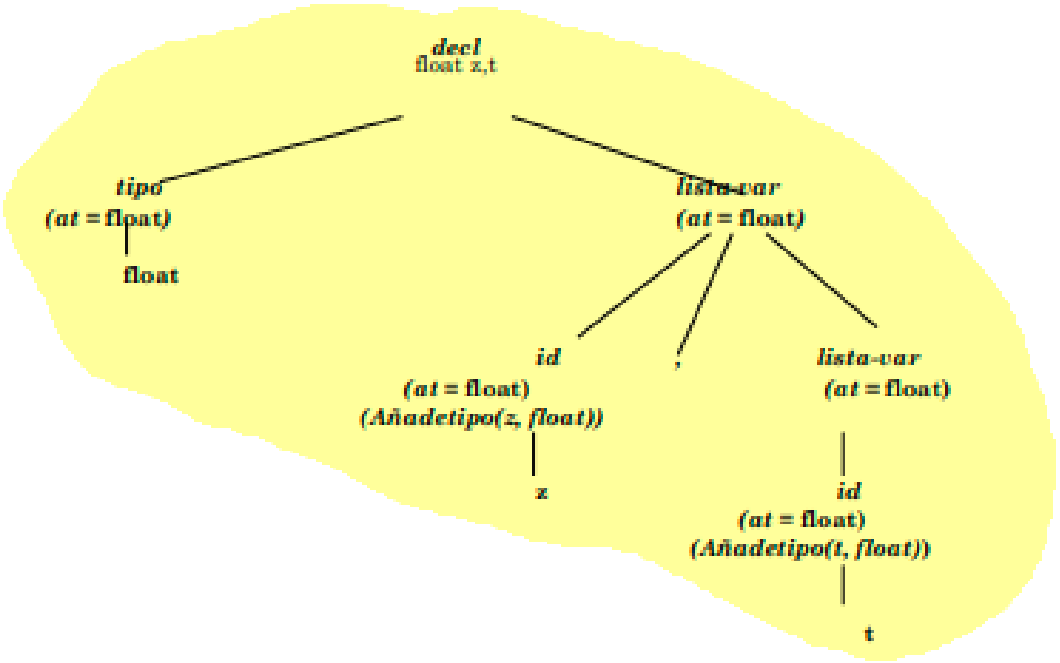
- Sirven para describir la sintaxis y la semántica estática formalmente.
- Son sensibles al contexto.
- Son utilizadas por los compiladores.

Funcionamiento de la Gramática de Atributos

- Asocia información a las construcciones del lenguaje a través de **atributos** que están asociados con los símbolos terminales y no terminales de la gramática, que sirven para la detección de errores.
- Los valores de los **Atributos** se obtienen mediante **ecuaciones o reglas semánticas** que están asociadas a las **producciones gramaticales.**
- Las **reglas sintácticas (producciones)** son similares a **BNF.**
- Las **reglas semánticas (ecuaciones)** permiten **detectar errores y obtener valores de atributos.**
- Un **Atributo** puede ser: el valor de una variable, el tipo de una variable o expresión, lugar que ocupa una variable en la memoria, dígitos significativos de un número, etc.
- Se suelen **expresar** en **forma tabular** para obtener el valor del **atributo.**



- Mira las **Reglas (simil BNF/EBNF)** y busca atributos para **terminales y no terminales,** si encuentra el **atributo** debe llegar a obtener su **valor**, esto lo logra **generando ecuaciones, usa la tabla** donde machea si **encuentra la producción/regla y del otro lado están ecuaciones que me permiten llegar a los atributos.**
- De la **ejecución de las ecuaciones:**
    - Se i**ngresan símbolos a la tabla de símbolos.**
    - Detectar y brindar **mensajes de error.**
    - Detecta dos **variables iguales.**
    - Controla **tipo y variables de igual tipo.**
    - Controla **reglas específicas del lenguaje.**
    - **Genera un código para el siguiente paso.**
- Parecido a las gramáticas **BNF y EBNF,** las **Gramáticas de Atributos** usan **Árboles sintácticos atribuidos.**



Semántica Dinámica

- **Describe** el **significado de ejecutar** las diferentes **construcciones** del **lenguaje de programación.**
- Su efecto se ve durante la **ejecución** del programa.
- Los programas sólo se pueden ejecutar si son correctos en la **sintaxis** y **semántica estática.**
- **No es fácil** escribirla.
- **No existen herramientas estándar** que sean fáciles y claras como los **diagramas sintácticos y BNF.**
- **Es complejo** describir la **relación entre entrada y salida del programa, cómo se ejecutará en cierta plataforma, etc.**

Soluciones más utilizadas

- Sirven para **comprobar la ejecución, la exactitud de un lenguaje, comparar funcionalidades de distintos programas.**
- **Se pueden usar combinados,** no todos sirven para todos los tipos de lenguajes de programación.
- Se dividen en:
    - **Formales y Complejas:** Semántica Axiomática y Semántica Denotacional.
    - **No Formal:** Semántica Operacional.

Semántica Axiomática

- Considera al **programa** como **"una máquina de estados"** donde cada instrucción provoca un **cambio de estado**.
- Se parte de un **axioma** (verdad) que sirve para **verificar "estados y condiciones"** a probar.
- Se desarrolló para **probar la corrección de los programas.**
- Un **estado** se describe con un **predicado**.
- El **predicado** describe los **valores** de las **variables** en ese **estado**.
- Existe un **estado anterior y un estado posterior** a la ejecución del constructor.

Semántica Denotacional

- Se basa en la **teoría de funciones recursivas y modelos matemáticos**, es más **exacto** para obtener y verificar resultados, pero es más **difícil** de leer.
- Describe los **estados** a través de **funciones** (recursivas).
- Define una **correspondencia** entre los **constructores sintácticos y sus significados.**
- **Lo que hace es buscar funciones que se aproximen a las producciones sintácticas.**

Semántica Operacional

- El **significado de un programa** se describe mediante **otro lenguaje de bajo nivel implementado sobre una máquina abstracta.**
- Cuando se **ejecuta** una **sentencia** del **lenguaje de programación** los cambios de **estado** de la **máquina abstracta definen su significado.**
- **Es un método informal** porque **se basa en otro lenguaje de bajo nivel y puede llevar a errores.**

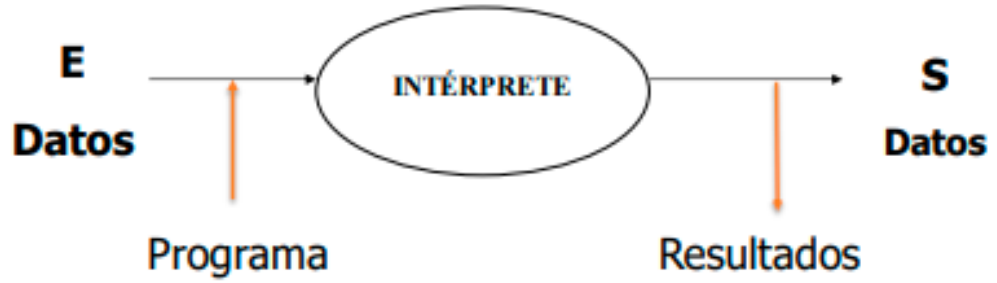
Procesamiento de un Programa

- Al comienzo se programaba en c**ódigo de máquina (0's y 1's)**, esto era muy **complejo** y tenía varios **errores** así que se necesitaba algún tipo de solución. Así surge el **Lenguaje Ensamblador** que utilizaba los llamados **códigos mnemotécnicos** para las **instrucciones** de código, la **desventaja** era que **cada máquina tenía su propio set de instrucciones** y por lo tanto, los **programas** eran **imposibles de intercambiar** entre distintas máquinas o familias de procesadores. Esto lleva al desarrollo de los **lenguajes de alto nivel**, ya que estos nos permitían poder **abstraernos** del **Lenguaje Ensamblador**, pero aun así se necesitaba algún tipo de **programa traductor** que

genere el **código en bajo nivel** de lo que escribimos en los **lenguajes de alto nivel** para que nuestros programas puedan ser **ejecutados**, para ello surgieron varias **alternativas**.

Interpretación

- Existe un **programa** que está **escrito** en un **lenguaje de programación interpretado** y hay un **programa** llamado **Intérprete** que realiza la **traducción** de ese **lenguaje interpretado en el momento de ejecución.**
- El **proceso** que realiza cuando se ejecuta sobre **cada una de las sentencias del programa** es:
    1. Leer.
    2. Analizar.
    3. Decodificar.
    4. Ejecutar.
- **Solo pasa por ciertas instrucciones del código**, esto está determinado por el flujo del programa y las decisiones que se tomen.
- Cada vez que vuelvo a ejecutar el programa se repite **toda la secuencia.**
- El **Intérprete** cuenta con una **serie de herramientas** para la **traducción** a lenguaje de máquina:
    - Por cada posible **acción** hay un **subprograma** en **lenguaje de máquina** que ejecuta esa acción.
    - La **interpretación** se realiza llamando a estos **subprogramas** en la **secuencia adecuada** hasta **generar el resultado de la ejecución.**



Compilación

- Existe un **programa** que está **escrito** en un **lenguaje de alto nivel no interpretado** y hay un **programa** llamado **Compilador** que realiza la **traducción** de ese **lenguaje no interpretado/lenguaje fuente a lenguaje de máquina.**
- Se **traduce/compila antes** de **ejecución**.
- **Pasa por todas las instrucciones** antes de la ejecución.
- **El código que se genera** se **guarda** y se puede **reusar** ya **compilado**.
- La **compilación** implica varias **etapas (Etapa de Análisis Y Etapa de Síntesis).**

- Luego de la compilación va a generar:
    - O un **lenguaje objeto** que es generalmente el **ejecutable (en lenguaje de máquina).**
    - O un **lenguaje de nivel intermedio (lenguaje ensamblador).**

Etapa de Análisis

- Vinculada con el código fuente.

Análisis Léxico (Scanner)

- Es un **proceso** que lleva **tiempo**.
- Hace el **análisis** a **nivel de palabra (LEXEMA) .**
- **Divide** el **programa** en sus **elementos/categorías**: identificadores, delimitadores, símbolos especiales, operadores, números, palabras clave, palabras reservadas, comentarios, etc.
- **Analiza** el **tipo** de cada uno para ver si son **TOKENS válidos.**
- **Filtra comentarios y separadores.**
- Lleva una **tabla** para la especificación del analizador léxico. Incluye cada **categoría**, el **conjunto de atributos** y **acciones asociadas.**
- Pone los **identificadores** en la **tabla de símbolos**. Reemplaza cada símbolo por su entrada en la tabla.
- Genera **errores** si la entrada **no coincide** con ninguna categoría léxica.
- El **resultado** de este **paso** será el **descubrimiento** de los í**tems léxicos o tokens y detección de errores**.

Análisis Sintáctico (Parser)

- El **análisis** se **realiza** a **nivel de sentencia/estructuras.**
- **Usa** los **tokens** del **analizador léxico.**
- **Tiene como objetivo encontrar las estructuras presentes en su entrada.**
- Estas **estructuras** se pueden **representar** mediante **el árbol de análisis sintáctico.**
- Se **identifican** las **estructuras** de las sentencias, declaraciones, expresiones, etc. ayudándose con los **tokens**.
- Se **alterna/interactúa** con el **análisis léxico y análisis semántico.**
- Usan **técnicas** de **gramáticas formales.**

Análisis Semántico (Semántica estática)

- **Debe pasar antes bien por Scanner y Parser.**
- Es una de las **fases** más **importantes**.

- **Procesa** las **estructuras sintácticas** (reconocidas por el **analizador sintáctico**).
- Agrega información implícita.
- La estructura del código ejecutable continúa tomando forma.
- **Realiza la comprobación de tipos** (**aplica gramática de atributos**) .
- **Agrega a la tabla de símbolos los descriptores de tipos.**
- Realiza **comprobaciones de duplicados, problema de tipos, etc.**
- Realiza **comprobaciones de nombres** (todas las variables deben estar declaradas).

Etapa de Síntesis

- Vinculada a características del código objeto, del hardware y la arquitectura.
- Construye el programa ejecutable y genera el código necesario.
- Se genera el módulo de carga. Programa objeto completo.
- Se realiza el proceso de optimización:
    - Es Optativo.
    - Los optimizadores de código (programas) pueden ser herramientas independientes, o estar incluidas en los compiladores e invocarse por medio de opciones de compilación.

Comparación entre Compilador e Intérprete

Por cómo se ejecuta

- **Intérprete**
    - Se utiliza en la **ejecución**.
    - **Ejecuta** el programa **línea por línea.**
    - Por donde pase **dependerá** de la **acción** del **usuario**, de la **entrada de datos** y/o de alguna **decisión** del programa.
    - **Siempre se debe tener el Programa Interprete.**
    - **El programa fuente será público** (necesito ambos).
- **Compilador**
    - Se utiliza **antes** de la **ejecución**.
    - Produce un **programa ejecutable** equivalente en **lenguaje objeto**.
    - **El programa fuente no será público.**

Por el orden de ejecución

- **Intérprete**
    - Sigue el **orden lógico** de ejecución (no necesariamente recorre todo el código).
- **Compilador**
    - Sigue el **orden físico** de las sentencias (recorre todo).

Por el tiempo consumido de ejecución

- **Intérprete**
    - **Por cada sentencia que pasa** realiza el proceso de **decodificación** (lee, analiza y ejecuta), es decir, **es repetitivo.**

    - Si la sentencia está en un **proceso iterativo** (ej.: for/while), se realizará la tarea de decodificación **tantas veces como sea requerido.**
    - La **velocidad** de proceso se puede ver **afectada**.
- **Compilador**
    - Pasa por **todas las sentencias.**
    - **No repite lazos.**
    - **Traduce** todo de **una sola vez.**
    - Genera código objeto ya compilado.
    - La **velocidad** de compilar **dependerá del tamaño del código**.

Por la eficiencia posterior

- **Intérprete**
    - Más **lento** en **ejecución**.
- **Compilador**
    - Es más **rápido** ejecutar desde el punto de vista del **hardware** porque ya está en un **lenguaje de más bajo nivel**.
    - **Detecta** más **errores** al pasar por **todas las sentencias.**
    - Está listo para ser **ejecutado**. Ya **compilado** es más **eficiente**.
    - **Tarda** más en **compilar** porque se **verifica todo previamente**.

Por el espacio ocupado

- **Intérprete**
    - Al **no pasar** por todas las **sentencias**, ocupa menos en la memoria.
    - Cada sentencia se deja en la forma original y las instrucciones interpretadas necesarias para ejecutarlas se almacenan en los subprogramas del intérprete en memoria.
    - Las Tablas de símbolos, variables y otros se generan cuando se usan en forma dinámica.
- **Compilador**
    - Al pasar por todas las sentencias, ocupa más espacio en la memoria.
    - Una sentencia puede ocupar decenas o centenas de sentencias de máquina al pasar a código objeto.
    - Cosas cómo tablas de símbolos, variables, etc. se generan siempre se usen o no.
    - En general, termina ocupando más espacio.

Por la detección de errores

- **Intérprete**
    - Las **sentencias del código** fuente pueden ser **relacionadas directamente** con la **sentencia en ejecución** entonces **se puede ubicar donde se produjo el error.**
    - Es más **fácil detectarlos** y **corregirlos** por donde pasa la **ejecución**.
- **Compilador**
    - Es **casi imposible ubicar el error**, pobres en significado para el programador.
    - Se deben usar otras **técnicas (ej. Semántica Dinámica).**

Combinación de Técnicas de Traducción

Primero Interpreto y luego Compilo

- Se utiliza el intérprete en la etapa de desarrollo para facilitar el diagnóstico de errores.
- Con el programa validado se compila para generar un código objeto más eficiente.

Primero Compilo y luego Interpreto

- Se hace traducción a un código intermedio a bajo nivel que luego se interpretará.
- Sirve para generar código portable, es decir, código fácil de transferir a diferentes máquinas y con diferentes arquitecturas.

Clase 3

Semántica Operacional

- Es **fundamental** para diversos aspectos del **proceso de desarrollo de software**, como el **diseño** de **lenguajes de programación**, la **verificación de programas** y la **comprensión** de cómo se **ejecutan** los **programas** en un **nivel más bajo**.

Entidades con las que trabajan los lenguajes

ENTIDADES	***ATRIBUTOS***
Variables	Nombre, Tipo, Área de Memoria, etc.
Rutinas (Funciones o Procedimientos)	Nombre, Parámetros formales y reales, convención de pasaje de parámetros, etc.
Sentencias	Acción asociada.
Descriptor: Lugar (repositorio) donde se almacena la información de los atributos anteriores.	
Si el **lenguaje es compilado**, el **Descriptor** se empieza a llenar en el momento de compilación para luego en ejecución seguir llenándose.	Si el **lenguaje es interpretado**, el **Descriptor** se empieza a llenar en el momento de ejecución de manera dinámica.

Ligadura o Binding

- Es el **momento** en el que a un **atributo** de una **entidad** se le asocia un **valor determinado**.
- Es un **concepto central** en la **definición** de la **semántica** de los lenguajes de programación.

- Se puede dividir en 2 tipos: **Estática y Dinámica.**

Concepto de Estabilidad

- Hace referencia a si una vez que una **Ligadura** es **establecida**, es decir, a un atributo se le asoció un valor determinado, esta **Ligadura ¿Se puede modificar o es fija?**

Momento de Ligadura

- Algunos **ATRIBUTOS** pueden **ligarse** en el momento de la **definición del lenguaje**, otros en el momento de **implementación**, en **tiempo de traducción (compilación)**, y otros en el **tiempo de ejecución.**
    - ***Definición del Lenguaje:*** Se establece **la forma** de las **sentencias**, la **estructura** del **programa** y los **nombres** de los **tipos predefinidos**.
    - ***Implementación:*** Se establece el **set de valores** y su **representación** además de sus **operaciones**.
    - ***Compilación:*** Se establece la **asignación/redefinición** del **tipo** a las **variables**.
    - ***Ejecución:*** Se **enlazan** las **variables** con sus **valores** y con su **lugar de almacenamiento**.

Ligadura Estática

- Se **establece antes de la ejecución del programa**, ya sea en la **definición del lenguaje**, la **implementación** o en la **compilación**.
- En cuanto a su **Estabilidad**, este tipo de **Ligadura no se puede modificar.**

Ligadura Dinámica

- Se **establece** durante la **ejecución**.
- En cuanto a su **Estabilidad**, **se puede modificar durante la ejecución del programa** de acuerdo a alguna **regla específica del lenguaje.**
- Existe una **excepción** en cuanto a la **Estabilidad** con las **Constantes**, su **ligadura** se **produce en tiempo de ejecución** pero esta **no puede ser modificada luego de ser establecida.**

Variables

- Una **variable** es una **celda de memoria** que ocupa una **dirección**, a la cual mediante una **sentencia de asignación** se le puede **brindar** y **modificar** el **contenido** que **almacena**, este contenido tiene una alta probabilidad de **cambiar** a lo largo de la ejecución de un programa, a excepción de que hablemos de una **constante**.
- También podemos ver a una **variable** como una **abstracción**:
    - La **variable** en sí es una abstracción **de una celda de memoria**.
    - El **nombre** de la misma es una abstracción de su **dirección de memoria**.
    - La **sentencia de asignación** que utiliza es una abstracción de la **modificación destructiva del valor que almacena la celda.**

- En cuanto a los **atributos** de una **variable**, podemos verla como una **5-tupla**, la cual estaría conformada de la siguiente manera **Atributos <Nombre, Alcance, Tipo, L-Value, R-Value>.**

Nombre

- **String de caracteres que se usa para referenciar a la variable. (identificador).**
- Es introducido por una **sentencia de declaración**.
- La **longitud máxima varía según el lenguaje**.
- Los **caracteres aceptados** en el nombre se denominan **conectores**.
- Pueden ser **case sensitive** o no.

Alcance

- **Rango de instrucciones** en el que es **conocido el nombre de la variable**. (**visibilidad**).
- Las **instrucciones** del programa pueden **manipular** las **variables** a través de su **nombre** dentro de su **alcance**. **Afuera** de ese **alcance** son **invisibles**.
- Existen **reglas de alcance** para **ligar** el **nombre** de una **variable** a su **alcance (Ligadura por Alcance Estático o Léxico y Ligadura por Alcance Dinámico).**

Ligadura por Alcance Estático o Léxico

- Se define el **alcance** en términos de la **estructura léxica del programa.**
- Puede **ligarse estáticamente** a una declaración de **variables** (explícita o implícita) **examinando el texto del programa**, sin necesidad de **ejecutarlo**, es decir, **el alcance depende de donde se declare la variable dentro del código.**
- Adoptada por la mayoría de lenguajes.

Ligadura por 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 **encontrada** durante la ejecución, es decir, e**n vez de tener en cuenta la estructura, se van viendo las instrucciones anteriores hasta encontrar la declaración de la variable.**
- **Más fáciles de implementar.**
- **Poco claras y eficientes.**
- En cuanto a la programación. **Encontrar una declaración en el flujo de ejecución puede ser duro.** El código se hace **más difícil de leer y seguir**, sobre todo en **grandes programas** con cientos de sentencias es complejo.

Clasificación de variables por su alcance

- **Global:** Son todas las referencias a variables creadas en el programa principal.
- **Local:** Son todas las referencias a variables que se han creado dentro de una unidad (programa o subprograma).

- **No Local:** Son todas las referencias que se utilizan dentro del subprograma pero que no han sido creadas en el subprograma. (son externas a él).

Espacio de Nombres

- Zona separada **abstracta** del código donde se pueden **agrupar**, **declarar** y **definir objetos** (variables, funciones, identificador de tipo, clase, estructura, etc.).
- Ayuda a **evitar problemas** con **identificadores** con el mismo **nombre** en grandes programas, o cuando se usan bibliotecas externas para **evitar colisión de nombres**.
- Se les asigna un nombre o identificador propio.
- **Ayudan a resolver el Alcance dentro de ese espacio de nombres**.

Tipo

- **Conjunto de valores** que se le pueden asociar a una **variable** junto con el **conjunto de operaciones permitidas** para la misma.
- **El tipo de una variable ayuda a:**
    - **Proteger** a las variables de **operaciones no permitidas**.
    - **Chequear tipos**.
    - **Verificar** el uso **correcto** de las variables.
    - **Detectar errores** en forma temprana.
    - Mejorar la **confiabilidad del código**.
- Antes de que una variable pueda ser referenciada, debe ligarsele un tipo.
- Existen 3 clases de Tipos: **Predefinidos por el lenguaje, Definidos por el Usuario y los Tipos de Datos Abstractos.**
- En cuanto a los momentos de ligadura, puede ocurrir de dos formas: **Estático o Dinámico.**

Tipos Predefinidos por el Lenguaje

- Son los **tipos base** que están descritos en la **Definición del Lenguaje**.
- Cada uno tiene **valores y operaciones**.
- Los valores se **ligan** en la **implementación** a representación de máquina según la arquitectura.

Tipos Definidos por el Usuario

- **Permiten al programador**:
    - Definir **nuevos tipos** a partir de los **tipos predefinidos** y de los **constructores**.
    - Crear **abstracciones**, **encapsular lógica** y **datos**, **reutilizar código** y **mejorar la claridad** y **legibilidad del código**.
- Son **esenciales** para la **organización** y la **abstracción**.
- Son **fundamentales** para el **desarrollo de programas complejos** y para mantener un **código organizado** y **mantenible**.

Tipos de Datos Abstractos (TAD)

- Son **estructuras de datos** que representan a un **nuevo tipo abstracto** con un nombre que los identifica.

- Está **compuesto** por una **colección de operaciones definidas** (**rutinas**). Las rutinas son usadas para manipular los objetos de este nuevo tipo.
- **Cada TAD define un conjunto de operaciones permitidas**, pero **oculta los detalles de implementación** interna.
- **No hay ligadura por defecto**, el programador debe especificar la representación y las operaciones.

Momento de Ligadura - Estático

- El tipo se liga en **compilación** y **no puede ser cambiado en ejecución**.
- **La ligadura entre variable y tipo se hace con la declaración.**
- **El chequeo de tipo también será estático.**
- La ligadura puede ser realizada en forma:
    - ***Explícita*:** La ligadura se establece mediante una sentencia de declaración, la ventaja de esta ligadura reside en la claridad de los programas y en una mayor fiabilidad, porque cosas como errores ortográficos en nombres de variables pueden detectarse en tiempo de traducción.
    - ***Implícito*:** Si no fue declarada la ligadura se deduce por "reglas propias del lenguaje". Esto ocurre sin que el programador tenga que especificar explícitamente el tipo de datos de la variable.
    - ***Inferido:*** El tipo se deduce automáticamente de los tipos de sus componentes. Se basa en el contexto del código y en el valor asignado a la variable. Se realiza en la traducción.

Momento de Ligadura - Dinámico

- El tipo se liga a la variable en **ejecución** y **puede modificarse**.
- **Cambia cuando se le asigna un valor** mediante una sentencia de asignación (no declaración).
- El **costo de implementación** de la ligadura dinámica es **mayor**, sobre todo el tiempo de ejecución por comprobación de tipos, mantenimiento del Descriptor asociado a cada variable en el que se almacena el tipo actual, cambio en el tamaño de la memoria asociada a la variable, etc.)
- **Chequeo dinámico**.
- **Menor legibilidad y errores.**

Reglas de Conversión

- **Implícitas (Promoción de tipo):** En operaciones aritméticas, promueve al tipo más general.
- **Explícitas:** Funciones integradas de conversión de tipo explícitas, como int(), float(), str(), list(), etc..

L-Valor

- **Es el área de memoria ligada a la variable durante la ejecución.**
- Las **instrucciones** de un programa **acceden** a la **variable** por su **L-Valor**.

Tiempo de Vida

- **Periodo de tiempo en que la variable está alocada en memoria y el binding existe.**
- Es desde que se solicita hasta que se libera.

Alocación

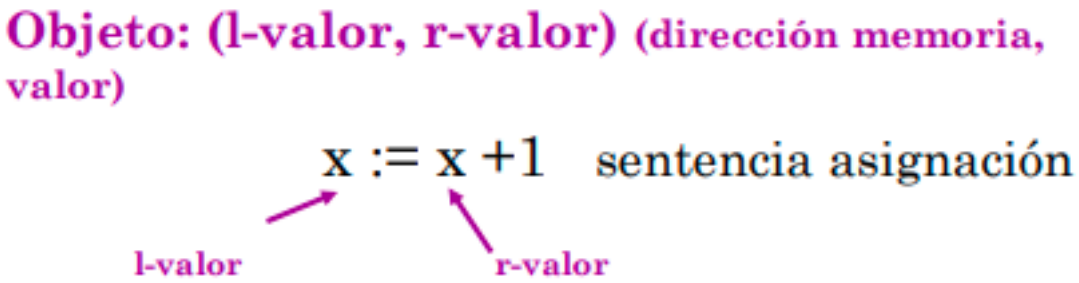
- **Momento en que se reserva la memoria para una variable.**
- Momento de la Alocación:
    - ***Estática:*** Se hace en **compilación** (antes de la ejecución) cuando se carga el programa en memoria en zona de datos y perdura hasta fin de la ejecución (sensible a la historia).
    - ***Dinámica:*** Se hace en tiempo de **ejecución**.
        - **Automática:** Cuando aparece una declaración en la ejecución.
        - **Explícita:** Requerida por el programador con la creación de una sentencia, a través de algún constructor.
    - ***Persistente:*** El tiempo de vida de los objetos no tiene relación con el tiempo de ejecución del programa. Persisten más allá de la memoria.

R-Valor

- **Es el valor codificado almacenado en la locación asociada a la variable (l-valor).**
- La codificación se interpreta de acuerdo con el tipo de la variable.

Momentos de Ligadura variable a valor

- ***Dinámico:*** El valor (r-valor) puede cambiar durante la ejecución con una asignación.
- ***Constantes:*** El valor (r-valor) no puede cambiar si se define como constante simbólica definida por el usuario.



Estrategias para la inicialización de variables

Inicialización por defecto

- **Las variables se inicializan con un valor por defecto**, por ejemplo los enteros en 0, los caracteres en blanco, etc.

Inicialización en la declaración

- **Las variables pueden inicializarse en el mismo momento que se declaran**, por ejemplo “int i = 0;”.

Ignorar el problema

- La **variable** toma como **valor inicial** lo que hay en **memoria** (la cadena de bits asociados al área de almacenamiento).
- Puede llevar a **errores** y **requiere chequeos adicionales**.

Puntero

- **Variable** que sirve para **señalar** la **posición de la memoria** en que se encuentra otro dato **almacenado** como **valor**, con la **dirección** de ese dato.

Alias

- Se da si hay variables que **comparten** un **objeto** en el **mismo entorno de referencia**, y **sus caminos de acceso conducen al mismo objeto.**
- **Si se modifica el objeto compartido vía un camino, se modifica para todos los demás.**
- **Ventaja**:
    - Compartir objetos se utiliza para mejorar la eficiencia.
- **Desventajas**:
    - Generar programas que sean difíciles de leer.
    - Generar errores porque el valor de una variable se puede modificar incluso cuando no se utiliza su nombre.

Sobrecarga

- Un **nombre** está **sobrecargado** si en un momento **referencia más de una entidad**.
- Debe estar permitido por el lenguaje.