

Autómatas II

Estrategia: Es una materia muy teórica en los exámenes por lo que una guía de estudio con la mayor información posible vista en el parcial, será lo mejor. Las tareas pueden rescatar puntos así como condenar la calificación

▼ Primer Parcial

▼ 09-08-2021

Profesor: Dr. Francisco Javier Ornelas Zapata

Actividades de entrega de trabajo y compartir trabajos por Aula Virtual

Clases por Teams

Se realizarán tres evaluaciones parciales con una ponderación de 25% c/u
Para las evaluaciones parciales se tendrá el 40% examen y 60% tareas, trabajos y participación en clase.
Un Proyecto final con la ponderación del 25%
Nota: Es necesario aprobar teoría y práctica por separado

▼ 10-08-2021

Evaluación diagnóstica

▼ 11-08-2021

No hubo clase

▼ 12-08-2021

Lenguaje

Es una facultad muy compleja del ser humano

El lenguaje es la capacidad que tiene el ser humano para expresarse y comunicarse a través de diversos sistemas de signos: orales, escritos o gestuales

La comunicación requiere de este sistema de signos para llegar al objetivo del entendimiento común

Diversos factores entran en juego cuando se realiza la comunicación por medio del lenguaje, se ponen en marcha funciones como la inteligencia y la memoria lingüística

El lenguaje constituye una de las características principales que separa al género humano de los animales

Lenguaje

- Es una capacidad innata que todos los individuos tienen, que comienza a desarrollarse a partir de la gestación y se establece definitivamente en la relación que mantiene el individuo con el mundo social en el que se desarrolla.
- Dominar esta capacidad de comunicación le permite a cada individuo expresar el mensaje que desea transmitir de un modo particular.
- Es importante mencionar que el resto de los animales también tiene sus propios métodos para comunicarse, pero, en este caso, hablaremos del "lenguaje" como característica humana.

Orígenes del lenguaje

- El lenguaje surge de la necesidad de los hombres de establecer relaciones sociales con el fin de la supervivencia de la especie humana.
- A lo largo del tiempo, fueron surgiendo diversos sistemas y lenguas que permitieron la exteriorización del lenguaje y la mejor comunicación entre los individuos.
- Sin embargo, las teorías actuales entienden que el lenguaje integra la constitución cerebral del ser humano y que la educación realiza únicamente la función de desarrollar en los individuos este impulso biológico que se encuentra latente.

Funciones del lenguaje

1. Función referencial o informativa: Comunicar determinada información
2. Función emotiva: Transmite un estado de ánimo
3. Función apelativa: Función para persuadir a una persona a hacer o no hacer algo
4. Función fática: Función para comprobar que el canal sigue abierto
5. Función estética: Utilizado para crear un tipo de belleza
6. Función lingüística: Se centra en el diálogo, se utiliza una lengua para hablar de ella misma

Características del lenguaje

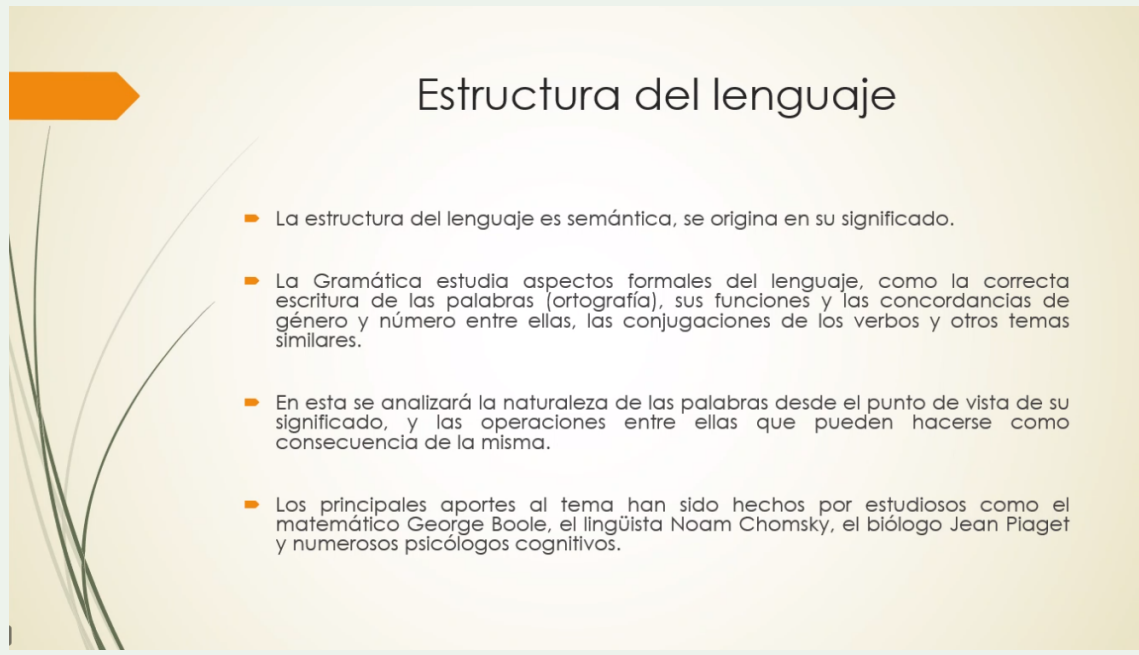
Es universal pero la educación ayuda a desarrollarlo

Permite intercambio de información entre emisor y receptor

Se vale de símbolos, señas

Puede ser verbal o no verbal

Estructura del lenguaje



Estructura del lenguaje

- La estructura del lenguaje es semántica, se origina en su significado.
- La Gramática estudia aspectos formales del lenguaje, como la correcta escritura de las palabras (ortografía), sus funciones y las concordancias de género y número entre ellas, las conjugaciones de los verbos y otros temas similares.
- En esta se analizará la naturaleza de las palabras desde el punto de vista de su significado, y las operaciones entre ellas que pueden hacerse como consecuencia de la misma.
- Los principales aportes al tema han sido hechos por estudiosos como el matemático George Boole, el lingüista Noam Chomsky, el biólogo Jean Piaget y numerosos psicólogos cognitivos.

Las reglas mediante las cuales los signos se pueden combinar para formar palabras y frases se llama sintaxis o gramática

La semántica estudia el significado de las frases al relacionar un contenido que se percibe en las frases

Tarea

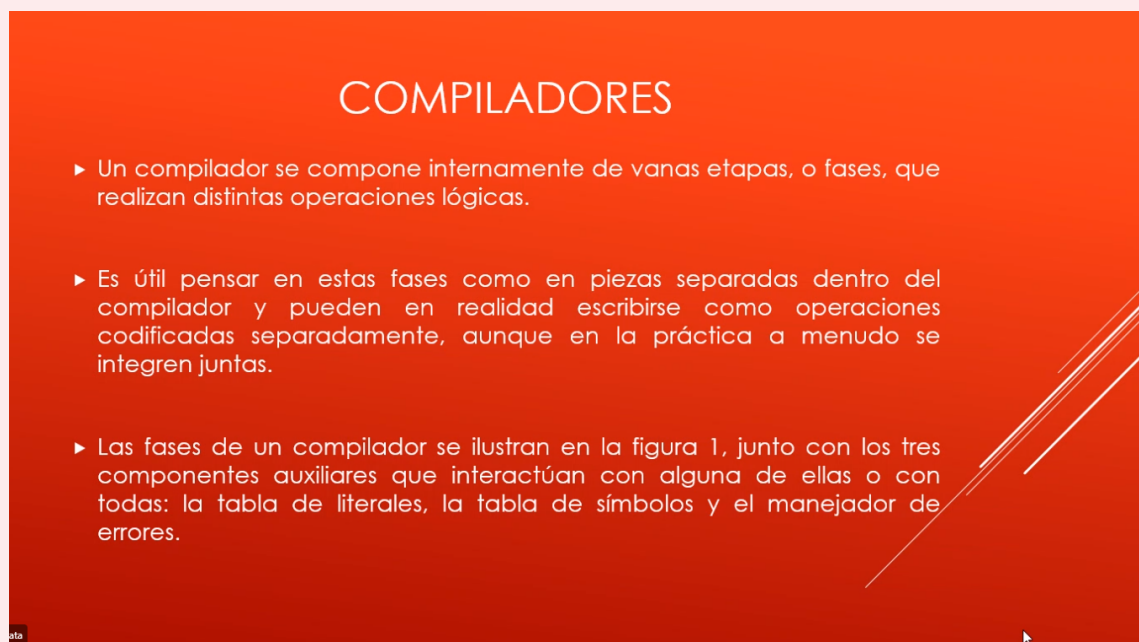
Identificar elementos del lenguaje y sus funciones

▼ 13-08-2021

Tarea: Investigación de reglas de sintaxis en lenguajes de programación

▼ 16-08-2021

Compiladores



COMPILADORES

- ▶ Un compilador se compone internamente de varias etapas, o fases, que realizan distintas operaciones lógicas.
- ▶ Es útil pensar en estas fases como en piezas separadas dentro del compilador y pueden en realidad escribirse como operaciones codificadas separadamente, aunque en la práctica a menudo se integren juntas.
- ▶ Las fases de un compilador se ilustran en la figura 1, junto con los tres componentes auxiliares que interactúan con alguna de ellas o con todas: la tabla de literales, la tabla de símbolos y el manejador de errores.

COMPILADORES



► Figura 1

COMPILADORES

- A continuación se describen brevemente cada una de las fases.
- **ANALIZADOR LÉXICO O RASTREADOR (SCANNER)**
- Esta fase del compilador efectúa la lectura real del programa fuente, el cual generalmente está en la forma de un flujo de caracteres.
- El rastreador realiza lo que se conoce como análisis léxico: recolecta secuencias de caracteres en unidades significativas denominadas tokens, las cuales son como las palabras de un lenguaje natural, como el inglés.
- De este modo, se puede imaginar que un rastreador realiza una función similar al deletreo.

COMPILADORES

- ▶ Como ejemplo, considere la siguiente línea de código, que podría ser parte de un programa en C:

`a[Index] = 4 + 2`

- ▶ Este código contiene 12 caracteres diferentes de un espacio en blanco pero sólo 8 tokens:

- | | |
|---------|--------------------|
| ▶ a | identificador |
| ▶ [| corchete izquierdo |
| ▶ Index | identificador |
| ▶] | corchete derecho |
| ▶ = | asignación |
| ▶ 4 | número |
| ▶ + | signo más |
| ▶ 2 | número |

COMPILADORES

- ▶ Cada token se compone de uno o más caracteres que se reúnen en una unidad antes de que ocurra un procesamiento adicional.
- ▶ Un analizador léxico puede realizar otras funciones junto con la de reconocimiento de tokens.
- ▶ Por ejemplo, puede introducir identificadores en la tabla de símbolos, y puede introducir literales en la tabla de literales (las literales incluyen constantes numéricas tales como 3.1415926535 y cadenas de texto entrecomilladas como "¡Hola, mundo!").

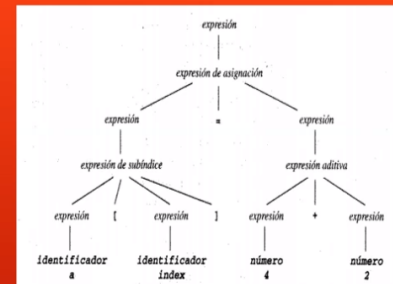
COMPILADORES

► ANALIZADOR SINTÁCTICO (PARSER)

- El analizador sintáctico recibe el código fuente en la forma de tokens proveniente del analizador léxico y realiza el análisis sintáctico, que determina la estructura del programa.
- Esto es semejante a realizar el análisis gramatical sobre una frase en un lenguaje natural.
- El análisis sintáctico determina los elementos estructurales del programa y sus relaciones.
- Los resultados del análisis sintáctico por lo regular se representan como un árbol de análisis gramatical o un árbol sintáctico.

COMPILADORES

- Como ejemplo, consideremos otra vez la línea de código en C que ya habíamos dado.
- Representa un elemento estructural denominado expresión, la cual es una expresión de asignación compuesta de una expresión con subíndice a la izquierda y una expresión aritmética entera a la derecha.
- Esta estructura se puede representar como un árbol de análisis gramatical de la forma siguiente



Se trata de la expresión:

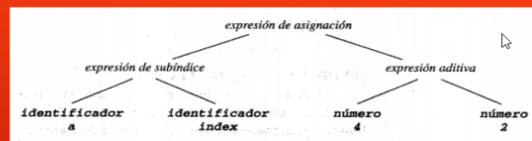
```
a[index]=4+2
```

COMPILADORES

- ▶ Como vemos los nodos internos del árbol de análisis gramatical están etiquetados con los nombres de las estructuras que representan y que las hojas del árbol representan la secuencia de tokens de la entrada.
- ▶ Un árbol de análisis gramatical es un auxiliar útil para visualizar la sintaxis de un programa o de un elemento de programa, pero no es eficaz en su representación de esa estructura.
- ▶ Los analizadores sintácticos tienden a generar un árbol sintáctico en su lugar, el cual es una condensación de la información contenida en el árbol de análisis gramatical.

COMPILADORES

- ▶ Un árbol sintáctico abstracto para nuestro ejemplo de una expresión de asignación en C es el siguiente:



- ▶ Como se ve, en el árbol sintáctico muchos de los nodos han desaparecido (incluyendo los nodos de tokens).
- ▶ Por ejemplo, si sabemos que una expresión es una operación de subíndice, entonces ya no será necesario mantener los corchetes que representan esta operación en la entrada original.

COMPILADORES

► ANALIZADOR SEMÁNTICO

- La semántica de un programa es su "significado", en oposición a su sintaxis, o estructura.
- La semántica de un programa determina su comportamiento durante el tiempo de ejecución, pero la mayoría de los lenguajes de programación tienen características que se pueden determinar antes de la ejecución e incluso no se pueden expresar de manera adecuada como sintaxis y analizarse mediante el analizador sintáctico.
- Se hace referencia a tales características como semántica estática, y el análisis de tal semántica es la tarea del analizador semántico (la semántica "dinámica" de un programa, es decir, aquellas propiedades del programa que solamente se pueden determinar al ejecutarlo, no se pueden determinar mediante un compilador porque éste no ejecuta el programa).

COMPILADORES

- Las características típicas de la semántica estática en los lenguajes de programación comunes incluyen las declaraciones y la verificación de tipos.
- Las partes extra de la información (como los tipos de datos) que se calculan mediante el analizador semántico se llaman atributos y con frecuencia se agregan al árbol como anotaciones o "decoraciones" (los atributos también se pueden introducir en la tabla de símbolos).

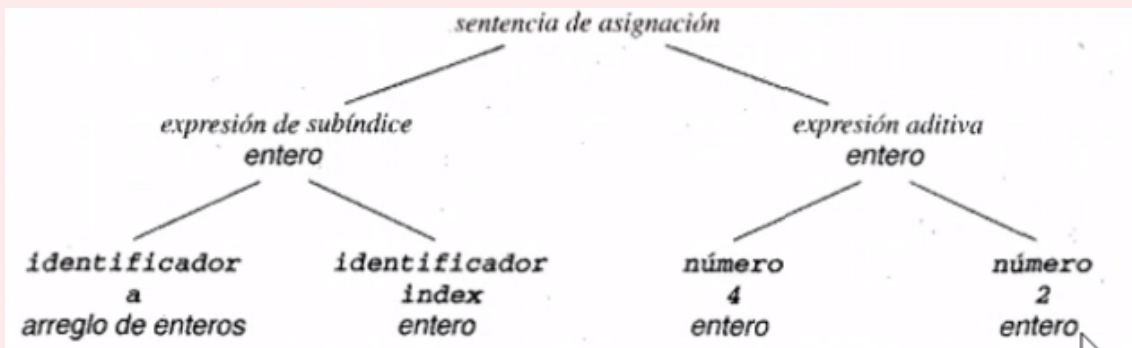
COMPILADORES

- En nuestro ejemplo de ejecución de la expresión en C

`a[index] = 4 + 2`

- la información de tipo específica que se tendría que obtener antes del análisis de esta línea sería que `a` sea un arreglo de valores enteros con subíndices proveniente de un subintervalo de los enteros y que `index` sea una variable entera.
- Entonces el analizador semántico registraría el árbol sintáctico con los tipos de todas las subexpresiones y posteriormente verificaría que la asignación tuviera sentido para estos tipos, y declararía un error de correspondencia de tipo si no fuera así.

En el resultado del análisis semántico en el árbol sintáctico podría representarse por el siguiente árbol con anotaciones:



COMPILADORES

► OPTIMIZADOR DE CÓDIGO FUENTE

- Los compiladores a menudo incluyen varias etapas para el mejoramiento, u optimización, del código.
- El punto más anticipado en el que la mayoría de las etapas de optimización se pueden realizar es precisamente después del análisis semántico, y puede haber posibilidades para el mejoramiento del código que dependerán sólo del código fuente.
- Indicamos esta posibilidad al proporcionar esta operación como una fase por separado en el de compilación.
- Los compiladores individuales muestran una amplia variación no sólo en los tipos de optimizaciones realizadas sino también en la colocación de las fases de optimización.

ata

COMPILADORES

- Muchas optimizaciones se pueden efectuar directamente sobre el árbol, pero en varios casos es más fácil optimizar una forma linealizada del árbol que esté más cercana al código ensamblador.
- Existen muchas variedades diferentes de tal código, pero una elección estándar es el código en tres direcciones, denominado así porque contiene las direcciones de (y hasta) tres localidades en memoria.
- Otra selección popular es el código P, el cual se ha utilizado en muchos compiladores de Pascal.

ata

COMPILADORES

- ▶ En nuestro ejemplo el código en tres direcciones para la expresión original en C podría parecerse a esto:

```
t = 4 + 2  
a [index] = t
```

- ▶ Advierta el uso de una variable temporal adicional t para almacenar el resultado intermedio de la suma.
- ▶ Ahora el optimizador mejoraría este código en dos etapas, en primer lugar calculando el resultado de la suma:

```
t = 6  
a [index] = t
```

COMPILADORES

- ▶ En la figura 1 indicamos la posibilidad de que el optimizador del código fuente pueda emplear código en tres direcciones al referirnos a su salida como a un código intermedio.
- ▶ Históricamente, el código intermedio ha hecho referencia a una forma de representación de código intermedia entre el código fuente y el código objeto, tal como el código de tres direcciones o una representación lineal semejante.
- ▶ Sin embargo, también puede referirse de manera más general a cualquier representación interna para el código fuente utilizado por el compilador.
- ▶ En este sentido, también se puede hacer referencia al árbol sintáctico como un código intermedio y efectivamente el optimizador del código fuente podría continuar el uso de esta representación en su salida.
- ▶ En ocasiones este sentido más general se indica al hacer referencia al código intermedio como representación intermedia, o RI.

▼ 17-08-2021

En la clase pasada vimos las fases:

1. Analizador léxico o rastreador (scanner)
2. Analizador sintáctico (parser)
3. Analizador semántico

4. Optimizador de código fuente

Ahora las nuevas etapas son:

5. Generador de código

Toma el código intermedio o RI y genera el código para la máquina objeto

Es en esta fase de la compilación en la que las propiedades de la máquina objeto se convierten en el factor principal

No sólo es necesario emplear instrucciones que existan en la máquina objeto, sino que las decisiones respecto a la representación de los datos desempeñarán ahora también un papel principal, tal como cuántos bytes o palabras de memoria ocuparán las variables de tipos de datos enteros y de punto flotante

Una posible secuencia de código muestra para la expresión dada podría ser:

```
MOV    R0, index    ;; valor de índice -> R0
MUL     R0, 2        ;; doble valor en R0
MOV     R1, &a       ;; dirección de a -> R1
ADD     R1, R0       ;; sumar R0 a R1
MOV     *R1, 6       ;; constante 6 -> dirección en R1
```

Hemos usado una convención propia de C para direccionar modos, de modo que &a es la dirección de a y que *R1 significa direccionamiento indirecto del registro

Partimos del supuesto de que la máquina realiza direccionamiento de byte y que los enteros ocupan dos bytes de memoria

6. Optimizador de código objeto

El compilador intenta mejorar el código objeto generado por el generador de código

Incluyen la selección de modos de direccionamiento para mejorar el rendimiento, reemplazando las instrucciones lentas por otras rápidas eliminando operaciones redundantes o innecesarias

► Con esto se completa una breve descripción de las fases de un compilador, pero se enfatiza que esta descripción sólo es esquemática y no necesariamente representa la organización real de un compilador trabajando, regularmente los compiladores muestran una amplia variación en sus detalles de organización, pero no obstante, las fases descritas están presentes de alguna forma en casi todos los compiladores.

▼ **18-08-2021**

Actividad de hacer un compilador en equipos

▼ **19-08-2021**

Trabajar en el compilador

▼ **20-08-2021**

Trabajar en el compilador

▼ **23-08-2021**

Procesamiento del Lenguaje Natural

Tarea investigación del PLN qué es, cómo ha evolucionado, principales áreas relacionado, principales áreas en uso

▼ **24-08-2021**

No hubo clase

▼ **25-08-2021**

Procesamiento del lenguaje natural

Tareas y aplicaciones del PLN

El recurso más importante que posee la raza humana es el conocimiento, es decir, la información

Durante la historia de la humanidad, la mayor parte del conocimiento se ha comunicado, guardado y manejado en la forma de lenguaje natural

El conocimiento sigue existiendo y creándose en forma de documentos, libros, artículos, aún cuando la mayoría se guarde en formato digital

Éste es precisamente el gran avance, que las computadoras sean una herramienta que ayuda enormemente en el procesamiento del conocimiento

Lo que es conocimiento para nosotros no lo es para las computadoras, para ellas son sólo archivos, secuencias de caracteres

Una computadora no puede buscar respuestas a las preguntas de un texto, ni hacer inferencias lógicas sobre su contenido, generalizarlo o resumirlo, es decir todo lo que las personas podemos hacer con un texto

- Para resolver esta situación se dedica mucho esfuerzo, sobre todo en los países más desarrollados del mundo, al desarrollo de la ciencia que se encarga de habilitar a las computadoras para entender un texto.
- Esta ciencia, en función del enfoque práctico o teórico, del grado en el que se espera lograr la comprensión y de otros aspectos, tiene varios nombres: procesamiento de lenguaje natural, procesamiento de texto, tecnologías de lenguaje, lingüística computacional.
- En todo caso, se trata de procesar el texto por su sentido y no como un archivo binario.

El esquema general de la mayoría de los sistemas y métodos es el siguiente:

1. El texto no se procesa directamente sino se transforma en una representación formal que preserva sus características relevantes para la tarea o el método específico
2. El programa principal manipula esta representación, transformándola según la tarea, buscando en ella las subestructuras necesarias
3. Si es necesario, los cambios hecho a la representación formal se transforman en lenguaje natural

- Para convertir a la computadora en un verdadero ayudante en el procesamiento de textos, se necesita pasar un largo camino de aprendizaje de la estructura de textos y de su formalización.
- Pero si es tan largo el camino, ¿existe una razón práctica para trabajar en esta área ahora? Sí, existe, porque con cada paso obtenemos herramientas de gran valor práctico que ayudan en nuestras tareas cotidianas.
- ¿Para qué sirve el procesamiento automático de lenguaje natural? En la práctica, se puede emplear en un rango de tareas que va desde situaciones muy simples a situaciones muy complejas.

Este tipo de aplicaciones quizá es conocido hoy en día por toda la gente que ha usado la computadora al menos una vez

Sólo nos interesan las herramientas que emplean el procesamiento complejo de texto y requieren conocimiento lingüístico

Ayuda en preparación de textos

Guiones: Tarea para determinar lugares donde las palabras pueden romperse para generar una nueva línea

Puede romperse la palabra como mara-villoso o maravillo-so pero no maravil-los o maravillos-o

Se requiere información profunda

Por ejemplo se debe saber cuáles son prefijo y raíz de una palabra:

Su-bir y sub-urbano, pero no sub-ir o su-burbano

Ortografía: Averiguar si una palabra está escrita correctamente o con un error

Tarea más difícil que la de los guiones

Como no es posible saber literalmente todas las palabras, se necesita saber en primer lugar las formas de las palabras

- Pero para detectar los errores de ortografía, o simplemente de escritura, se debe considerar el contexto de la palabra de una manera a veces bastante compleja.
- Ejemplos:
 - Sé que piensen en el futuro.
 - El terremoto causó una gran hola en el mar.
 - Se iba a cazar con su novia en la iglesia bautista.
 -
- Para detectar este tipo de errores, la computadora necesita, incluso, entender hasta cierto grado el sentido del texto.

▼ 26-08-2021

Gramática: Los correctores de gramática detectan las estructuras incorrectas en las oraciones, aunque todas las palabras en la oración estén bien escritas, por ejemplo:

- Quiero que viene mañana
- El maestro de matemáticas, se fue
- Me gusta la idea ir a Europa
- Fuera magnífico si él venía a la fiesta

El problema para detectar errores de este tipo es que hay una gran variedad de estructuras permitidas

Estilo. Una tarea más complicada consiste en detectar en el texto los problemas de las palabras correctamente escritas y las oraciones correctamente estructuradas, pero poco legibles, ambiguas, mal estructuradas, inconsistentes en el uso de palabras de diferentes estilos

En un texto científico no se debe usar "caló"

Una carta a un amigo no se hace con oraciones largas o con palabras científicas

Hay que emplear un análisis profundo

Hechos y coherencia lógica. Probablemente en el futuro los correctores de texto serán capaces de encontrar errores como éstos:

Cuando voy a Inglaterra, quiero visitar París, la capital de este país

Primero vino Pedro y después José; ya que José llegó antes de Pedro, tomó el mejor asiento

Antes la pregunta era "¿Cómo puedo obtener la información?" y ahora la pregunta principal es "¿Dónde puedo encontrar la información?"

La aplicación de PLN más obvia y quizá la más importante en la actualidad, es la búsqueda de información

Internet y las bibliotecas digitales contienen una cantidad enorme de conocimiento que puede dar respuestas a muchísimas preguntas que tenemos

Rara vez se trata de decidir cuáles documentos son relevantes para la petición del usuario y cuáles no

La tarea consiste en medir el grado de relevancia

Las ideas más usadas son los modelos probabilísticos y los procedimientos iterativos e interactivos, tratando de adivinar qué necesita el usuario y cuáles documentos le sirven

Si los métodos analizan las palabras por separado, pero no el sentido del texto son muy limitados

Coincidencia de las formas morfológicas de palabras: Buscando pensar, encontrar piénsalo

Es bastante simple de resolver en el lenguaje inglés, pero para el español, se convierte en un problema moderadamente serio, debido a la gran variedad de las formas de las palabras en el español

Coincidencia de los sinónimos, conceptos más generales y más específicos: Buscando cerdo encontrar puerco, mascota, animal

Tomar en cuenta las relaciones entre las palabras en la petición del usuario y el documento regresado

▼ 27-08-2021

No hubo clase

▼ 30-08-2021

Continuación del PLN

- Los programadores utilizaban tarjetas perforadas para comunicarse con las primeras computadoras hace 70 años. Este proceso manual y arduo era entendido por un número relativamente reducido de personas.
- Actualmente se puede decir, “Alexa, me gusta esta canción” y el dispositivo que reproduce música, bajará el volumen y le responderá, “OK. Evaluación guardada”, en una voz similar a la humana, luego adapta su algoritmo para reproducir esa canción la próxima vez que escuche esa lista musical.

Estructuración de una fuente de datos altamente no estructurada

- El lenguaje humano es increíblemente complejo y diverso.
 - Nos expresamos de maneras infinitas, verbalmente y por escrito.
 - Existen cientos de lenguajes y dialectos, pero en cada uno existe un conjunto único de reglas gramáticas y de sintaxis, términos y palabras coloquiales.
 - Cuando escribimos, a menudo cometemos errores ortográficos o abreviamos palabras, o bien omitimos signos de puntuación.
 - Cuando hablamos, tenemos acentos regionales, y mascullamos, tartamudeamos o tomamos palabras prestadas de otros idiomas.
-
- Aunque el aprendizaje supervisado y no supervisado, se utilizan ampliamente para modelar el lenguaje humano, se necesitan también entendimiento sintáctico y semántico, así como conocimientos de dominio que no están necesariamente presentes en estos métodos de machine learning.
 - PLN es importante porque ayuda a resolver la ambigüedad del lenguaje y agrega estructura numérica útil a los datos para muchas aplicaciones industriales, como el reconocimiento del habla o la analítica de texto.

¿Qué puede hacer la analítica de texto por su organización?

La analítica de texto es un tipo de procesamiento del lenguaje natural que convierte el texto en datos para su análisis. Aprenda cómo organizaciones de la banca, atención a la salud y ciencias de la vida, manufactura y gobierno utilizan la analítica de texto para ofrecer mejores experiencias a sus clientes, reducir el fraude y mejorar la convivencia social.

https://www.sas.com/es_mx/whitepapers/text-analytics-for-executives-109630.html

Detecte la septicemia antes de que se vuelva mortal

Dignity Health utiliza PLN y otros algoritmos avanzados para monitorear registros médicos electrónicos para detectar indicios de septicemia. Si la probabilidad de que un paciente presente septicemia es alta, el sistema envía una alarma a la enfermera o médico de guardia.

https://www.sas.com/es_mx/insights/articles/analytics/using-analytics-to-prevent-sepsis.html

▼ 31-08-2021

No hubo clase

▼ 01-09-2021

¿Cómo funciona PLN?

Separación de las piezas elementales del lenguaje

- El procesamiento del lenguaje natural incluye diferentes técnicas para interpretar el lenguaje humano, que van desde los métodos estadísticos y del aprendizaje basado en máquina hasta los enfoques basados en reglas y algoritmos.

- Se requiere una amplia variedad de métodos porque los datos basados en texto y en voz varían ampliamente, al igual que las aplicaciones prácticas.
- En términos generales, las tareas PLN dividen el lenguaje en piezas elementales más cortas, intentan entender las relaciones entre las piezas y exploran cómo funcionan las piezas juntas para crear significado.

Estas tareas implícitas se utilizan a menudo en recursos PLN de más alto nivel, como:

- Categorización de contenido. Un resumen del documento basado en la lingüística, incluyendo búsqueda e indización, alertas de contenido y detección de duplicación.
- Descubrimiento y modelado de temas. Capture con precisión el significado y temas en colecciones de texto, y aplique analítica avanzada a texto, como optimización y pronósticos.
- Sumarización de documentos. Generación automática de sinopsis de grandes cuerpos de texto.
- Traducción basada en máquina. Traducción automática de texto o habla de un idioma a otro.

En todos estos casos, la meta general es tomar texto crudo del lenguaje y aplicar la lingüística y algoritmos para transformar o enriquecer el texto de tal forma que provea un mayor valor.

SAS® Visual Text Analytics

¿Cómo puede encontrar respuestas en grandes volúmenes de datos textuales? Combinando el aprendizaje basado en máquina con el procesamiento del lenguaje natural y la analítica de texto. Averigüe cómo se pueden analizar sus datos no estructurados para identificar problemas, evaluar el sentimiento, detectar tendencias emergentes e identificar oportunidades ocultas.

https://www.sas.com/es_mx/software/visual-text-analytics.html

Métodos y aplicaciones PLN

Cómo entienden las computadoras los datos textuales

- El procesamiento del lenguaje natural va de la mano de la analítica de texto, la cual cuenta, agrupa y categoriza palabras para extraer estructura y significado de grandes volúmenes de contenido.
- La analítica de texto se utiliza para explorar contenido textual y derivar nuevas variables de texto crudo que se pueden visualizar, filtrar o utilizar como entradas para modelos predictivos u otros métodos estadísticos.

Ver Watson

▼ 02-09-2021

6 al 17 de septiembre para evaluación

▼ 03-09-2021

No hubo clase

▼ 06-09-2021

Continuación de análisis del lenguaje

Análisis del lenguaje

La estructura del lenguaje se analiza en cuatro niveles

“El niño corre”

- ANÁLISIS MORFOLÓGICO

- El análisis de las palabras para extraer raíces, unidades léxicas. Se identifican los componentes en su mínima expresión.

El - artículo

niño - sustantivo

corre - verbo

Análisis del lenguaje

- ANÁLISIS SINTÁCTICO

- El análisis de la estructura sintáctica de la frase mediante una gramática, es decir, el orden de los elementos identificados anteriormente.

Sujeto = artículo + sustantivo + verbo
El niño corre

Análisis del lenguaje

- ANÁLISIS SEMÁNTICO

- La extracción del significado (o posibles significados) de la frase

- ANÁLISIS PRAGMÁTICO

- El análisis de los significados más allá de los límites de la frase, por ejemplo, para determinar los antecedentes referenciales de los pronombres.

Análisis del lenguaje

- Una gramática independiente de contexto esta formada por:

- Un conjunto de símbolos terminales T (las palabras)
- Un conjunto de símbolos no terminales N (los constituyentes o categorías sintácticas)
- Un símbolo no terminal inicial S
- Un conjunto de reglas de producción, que indican las maneras en que se puede derivar una oración válida a partir del símbolo inicial
 - Estas reglas son de la forma $N \Rightarrow W$, donde $N \in N$ y $W \in (N \cup T)^*$

Análisis del lenguaje

- Ejemplo

S $\Rightarrow$ NP VP
NP $\Rightarrow$ DT N
| N
| NP PP
VP $\Rightarrow$ V NP
| V
| VP PP

PP $\Rightarrow$ P NP
| P
DT $\Rightarrow$ el | los
N $\Rightarrow$ hombre | amigos | café | leche
V $\Rightarrow$ toma | toman
P $\Rightarrow$ con | solo

Análisis del lenguaje

- Ejemplo

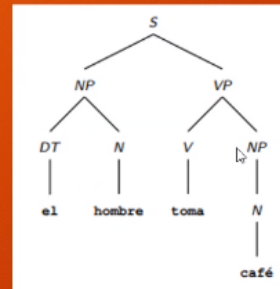
S $\Rightarrow$ NP VP
 $\Rightarrow$ DT N VP
 $\Rightarrow$ el N VP
 $\Rightarrow$ el hombre VP
 $\Rightarrow$ el hombre V NP
 $\Rightarrow$ el hombre toma NP
 $\Rightarrow$ el hombre toma N
 $\Rightarrow$ el hombre toma café

Podemos abreviar la derivación de la siguiente forma:

S $\Rightarrow^*$ el hombre toma café

Análisis del lenguaje

- La forma en que se ha realizado la derivación se puede capturar con un árbol de derivación sintáctica
- Ejemplo:



Análisis del lenguaje

- El lenguaje generado por una gramática G es el conjunto de frases s para las que existe una derivación:
$$L(G) = \{s \in T^* \mid S \Rightarrow^* s\}$$
- Formas de utilizar una gramática:
 - Para generar texto
 - Para analizar texto (parsing), obteniendo el árbol sintáctico

Análisis del lenguaje

- Existen numerosos algoritmos de parsing eficientes (compiladores)
- Las gramáticas independientes del contexto son muy útiles para procesar lenguajes formales (con pocos símbolos terminales y pocas reglas)

Análisis del lenguaje

- Los lenguajes naturales son mucho más expresivos que los lenguajes descritos por gramáticas independientes de contexto
- Concordancia morfológica: género, número, tiempos verbales, pronombres. Por ejemplo, $S \Rightarrow *$ el amigos toma café
 - Deberíamos tener un conjunto de reglas para frases en plural y otro para frases en singular.
 - El número de reglas aumenta exponencialmente si se quieren tener en cuenta todas las concordancias

Análisis del lenguaje

- Otro problema: ambigüedades sintáctica y semántica
 - La misma frase tiene distintos árboles de derivación sintáctica, aunque sólo uno de ellos es correcto a nivel semántico: Él toma café con leche
 - La misma frase tiene distintos árboles de derivación sintáctica, y ambas son correctas a nivel semántico: El toma café solo

Hacer equipos para un PLN básico

▼ 07-09-2021

No hubo clase

▼ 08-09-2021

No hubo clase dado que se realizó el examen de Redes I

Actividades en Aula

Examen lunes a las 8:00 con duración de una hora

▼ 09-09-2021

Tareas y ejercicios y aclaraciones sobre el examen

▼ 10-09-2021

No hubo clase

▼ 13-09-2021

No hubo clase debido a la realización del primer parcial

▼ Segundo Parcial

▼ 14-09-2021

Intérpretes

- Los compiladores, a diferencia de los intérpretes, transforman el programa a un programa equivalente en un código objeto (fase de compilación), y en un segundo paso generan los resultados a partir de los datos de entrada (fase de ejecución) Fig. 2.

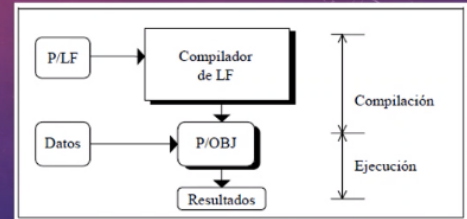


Fig. 2 Esquema general de un compilador

ESTRUCTURA DE UN INTERPRETE

A la hora de construir un intérprete es conveniente utilizar una Representación Interna (RI) del lenguaje fuente a analizar, de esta forma, la organización interna de la mayoría de los intérpretes se descompone en los módulos siguientes (Fig. 3):

- **Traductor a Representación Interna:** Toma como entrada el código del programa P en lenguaje fuente (LF), lo analiza y lo transforma a la representación interna correspondiente a dicho programa P.
- **Representación Interna (P/RI):** La representación interna debe ser consistente con el programa original. Entre los tipos de representación interna, los árboles sintácticos son los más utilizados y, si las características del lenguaje lo permiten, pueden utilizarse estructuras de pila para una mayor eficiencia.

ESTRUCTURA DE UN INTERPRETE

- **Tabla de símbolos:** Durante el proceso de traducción, es conveniente ir creando una tabla con información relativa a los símbolos que aparecen. La información a almacenar en dicha tabla de símbolos depende de la complejidad del lenguaje fuente. Se pueden almacenar etiquetas para instrucciones de salto, información sobre identificadores (nombre, tipo, línea en la que aparecen, etc.) o cualquier otro tipo de información que se necesite en la etapa de evaluación.
- **Evaluador de Representación Interna:** A partir de la Representación Interna anterior y de los datos de entrada, se llevan a cabo las acciones indicadas para obtener los resultados. Durante el proceso de evaluación es necesario contemplar la aparición de errores.

ESTRUCTURA DE UN INTERPRETE

- **Tratamiento de errores:** Durante el proceso de evaluación pueden aparecer diversos errores como desbordamiento de la pila, divisiones por cero, etc., que el intérprete debe contemplar.

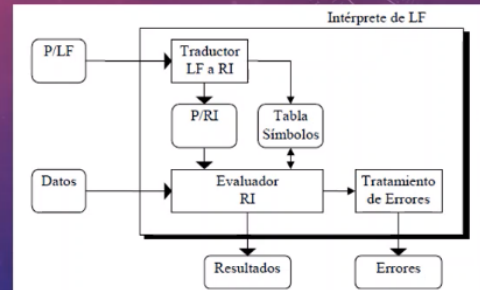


Fig. 3 Organización interna de un intérprete

ESTRUCTURA DE UN INTERPRETE

- Dependiendo de la complejidad del código a analizar, el intérprete puede contener módulos similares a los de un compilador tradicional: Análisis léxico, Sintáctico y Semántico.
- Durante la evaluación, el intérprete interactúa con los recursos del sistema como la memoria, discos, etc.
- Muchos sistemas interpretados liberan al programador del manejo explícito de memoria mediante técnicas de recolección de basura.
- A la hora de evaluar la representación interna, existen dos métodos fundamentales: la interpretación iterativa y la interpretación recursiva.

INTERPRETACIÓN ITERATIVA

- La interpretación iterativa es apropiada para lenguajes sencillos, donde se analiza y ejecuta cada expresión de forma directa, como podrían ser los códigos de máquinas abstractas o lenguajes de sentencias simples.
 - La interpretación consiste en un ciclo básico de búsqueda, análisis y ejecución de instrucciones.
- El esquema sería:
- ```
Inicializar
REPETIR
 Buscar siguiente Instrucción i
 SI encontrada ENTONCES
 Analizar i
 Ejecutar i
HASTA (que no haya más instrucciones)
```

## INTERPRETACIÓN RECURSIVA

Comúnmente, el diseño de nuevos lenguajes de programación se realiza en dos fases:

- Una primera fase de especificación semántica mediante la construcción de un intérprete prototipo que actúa como una especificación ejecutable y una segunda fase de implementación del compilador de dicho lenguaje.
  - Para la construcción de prototipos suele utilizarse un modelo de interpretación recursiva donde las sentencias pueden estar compuestas de otras sentencias y la ejecución de una sentencia puede lanzar la ejecución de otras sentencias de forma recursiva.
- Los intérpretes recursivos no son apropiados para aplicaciones prácticas debido a su ineficiencia y se utilizan únicamente como prototipo ejecutable del lenguaje.
  - El problema de especificar un lenguaje mediante un intérprete prototipo es decidir en qué lenguaje se implementa dicho intérprete.
  - Dicho lenguaje debe ser suficientemente expresivo y no ambiguo para definir claramente cómo funcionan las diferentes construcciones.

## VENTAJAS DE LA UTILIZACIÓN DE INTERPRETES

- En general, la utilización de compiladores permite construir programas más eficientes que los correspondientes interpretados, esto es debido a que durante la ejecución de código compilado no es necesario realizar complejos análisis (ya se hicieron en tiempo de compilación), además, un buen compilador es capaz de detectar errores y optimizar el código generado.
- Los intérpretes, por definición, realizan la fase de análisis y ejecución a la vez, lo cual imposibilita tales optimizaciones, por esta razón, los sistemas interpretados suelen ser menos eficientes que los compilados.

## VENTAJAS DE LA UTILIZACIÓN DE INTERPRETES

A continuación se enumeran una serie de ventajas de los sistemas interpretados:

- Los intérpretes, en general, son más sencillos de implementar, lo cual facilita el estudio de la corrección del intérprete y proporciona nuevas líneas de investigación como la generación automática de intérpretes a partir de las especificaciones semánticas del lenguaje.
- Proporcionan una mayor flexibilidad que permite modificar y ampliar características del lenguaje fuente, muchos lenguajes como Lisp, APL, Prolog, etc. surgieron en primer lugar como sistemas interpretados y posteriormente surgieron compiladores.
- No es necesario contener en memoria todo el código fuente, esto permite su utilización en sistemas de poca memoria o en entornos de red, en los que se puede obtener el código fuente a medida que se necesita.

## VENTAJAS DE LA UTILIZACIÓN DE INTERPRETES

- Facilitan la meta-programación: un programa puede manipular su propio código fuente a medida que se ejecuta. esto facilita la implementación de sistemas de aprendizaje automatizado y reflectividad.
- Aumentan la portabilidad del lenguaje: Para que el lenguaje interpretado funcione en otra máquina sólo es necesario que su intérprete funcione en dicha máquina, puesto que no existen etapas intermedias de compilación, los sistemas interpretados facilitan el desarrollo rápido de prototipos, potencian la utilización de sistemas interactivos y facilitan las tareas de depuración.

Investigar al menos dos lenguajes interpretados y dos lenguajes compilados

▼ 15-09-2021



No hubo clase

## ▼ 20-09-2021

Continuación de los intérpretes

### APLICACIONES DE LOS SISTEMAS BASADOS EN INTÉRPRETES

Los sistemas interpretados han tenido una gran importancia desde la aparición de la computadora, en la actualidad, la evolución del hardware abre nuevas posibilidades a los sistemas interpretados, la preocupación ya no es tanto la eficiencia como la capacidad de desarrollo rápido de nuevas aplicaciones.

Las principales aplicaciones podrían resumirse en:

**Intérpretes de Comandos:** Los sistemas operativos cuentan con intérpretes de comandos como el Korn-Shell, C-Shell, JCL, etc. Estos intérpretes toman un lenguaje fuente que puede incluir sentencias de control (bucles, condiciones, asignaciones, etc.) y ejecutan los diferentes comandos a medida que aparecen en el lenguaje.

**Lenguajes basados en Scripts (Scripting Languages):** diseñados como herramientas que sirvan de enlace entre diferentes sistemas o aplicaciones. Suelen ser interpretados con el fin de admitir una mayor flexibilidad a la hora de afrontar las peculiaridades de cada sistema. Podrían destacarse Perl, Tcl/Tk, JavaScript, WordBasic.

- **Entornos de Programación:** Existen ciertos lenguajes que contienen características que impiden su compilación o cuya compilación no es efectiva. Estos lenguajes suelen disponer de un complejo entorno de desarrollo interactivo con facilidades para la depuración de programas. Entre estos sistemas pueden destacarse los entornos de desarrollo para Lisp, Visual Basic, Smalltalk, etc.
- **Lenguajes de Propósito Específico:** Ciertos lenguajes incluyen sentencias que realizan tareas complejas en contextos específicos. Existe una gran variedad de aplicaciones en las que se utilizan este tipo de lenguajes como consultas de Bases de Datos, simulación, descripción de hardware, robótica, CAD/CAM, música, etc.
- **Sistemas en Tiempo Real:** Entornos que permiten modificar el código de una aplicación en tiempo de ejecución de forma interactiva.
- **Intérprete de Código Intermedio:** Una tendencia tradicional en el diseño de compiladores es la generación de un código intermedio para una máquina abstracta, por ejemplo, el P-Code de Pascal o los bytecodes de Java. El siguiente paso puede ser: generación del código objeto a partir del código intermedio para una máquina concreta, finalizando el proceso de compilación o interpretar dicho código intermedio en una máquina concreta. La tendencia habitual es definir un lenguaje intermedio independiente de una máquina concreta. Para ello, suele definirse una máquina virtual que contenga las instrucciones definidas por el lenguaje intermedio, permitiendo una mayor portabilidad. Un ejemplo sería la Máquina Virtual de Java, que es simulada en la mayoría de los visualizadores Web.

## TIPOS DE INTÉRPRETES

- A continuación se va a realizar una clasificación de los diferentes métodos de interpretación según la estructura interna del intérprete.
- Es conveniente observar que algunos métodos podrían considerarse híbridos, ya que mezclan los procesos de compilación e interpretación.

## TIPOS DE INTÉRPRETES

### Intérpretes puros

- Los intérpretes puros son los que analizan y ejecutan sentencia a sentencia todo el programa fuente.
- Siguen el modelo de interpretación iterativa y, por tanto, se utilizan principalmente para lenguajes sencillos.
- Los intérpretes puros se han venido utilizando desde la primera generación de computadoras al permitir la ejecución de largos programas en computadoras de memoria reducida, ya que sólo debían contener en memoria el intérprete y la sentencia a analizar y ejecutar en cada momento.
- El principal problema de este tipo de intérpretes es que si a mitad del programa fuente se producen errores, se debe de volver a comenzar el proceso.



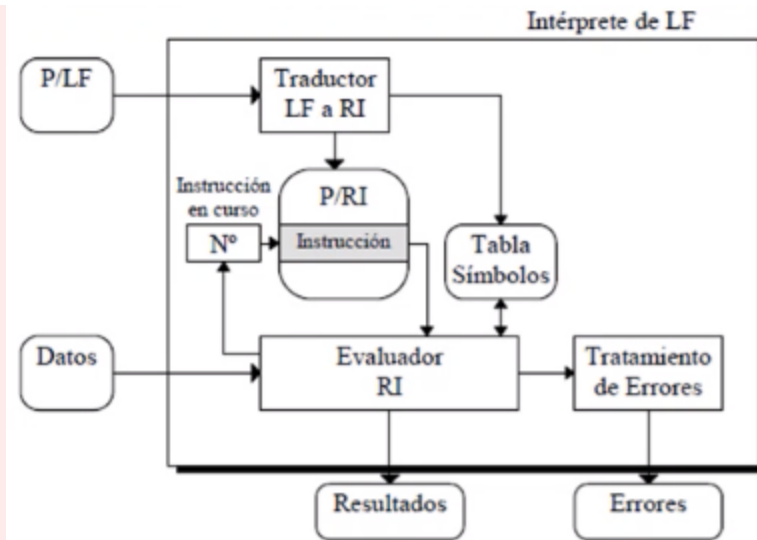


Figura 1. Esquema de un intérprete puro

## TIPOS DE INTÉRPRETES

- En la figura 1 se representa el esquema general de un intérprete puro.
- Se puede observar que el lenguaje fuente se traduce a una representación interna (texto o binaria) que puede ser almacenada en memoria o en disco.
- Esta representación interna tiene todas las instrucciones numeradas o colocadas consecutivamente en estructuras de tamaño fijo (por ejemplo un array o posiciones consecutivas de memoria, o un fichero binario de estructuras de tamaño fijo).
- Mientras se realiza este paso se puede construir la tabla de símbolos o etiquetas, que es una tabla que contiene una estructura donde están todas las etiquetas y su posición en el programa fuente (las etiquetas se utilizan tanto en las instrucciones de salto como en las llamadas a procedimientos y funciones).

## TIPOS DE INTÉRPRETES

### Intérpretes avanzados

- Los intérpretes avanzados o normales incorporan un paso previo de análisis de todo el programa fuente. Generando posteriormente un lenguaje intermedio que es ejecutado por ellos mismos.
- De esta forma en caso de errores sintácticos no pasan de la fase de análisis.
- Se utilizan para lenguajes más avanzados que los intérpretes puros, ya que permiten realizar un análisis más detallado del programa fuente (comprobación de tipos, optimización de instrucciones, etc.)

## TIPOS DE INTÉRPRETES

### Intérpretes incrementales

- Existen ciertos lenguajes que, por sus características, no se pueden compilar directamente, la razón es que pueden manejar objetos o funciones que no son conocidos en tiempo de compilación, ya que se crean dinámicamente en tiempo en ejecución.
  - Entre estos lenguajes, pueden considerarse Smalltalk, Lisp o Prolog.
  - Con el propósito de obtener una mayor eficiencia que en la interpretación simple, se diseñan compiladores incrementales.
- 
- La idea es compilar aquellas partes estáticas del programa en lenguaje fuente, marcando como dinámicas las que no puedan compilarse, posteriormente, en tiempo de ejecución, el sistema podrá compilar algunas partes dinámicas o recompilar partes dinámicas que hayan sido modificadas.
  - Estos sistemas no producen un código objeto independiente, sino que acompañan el sistema que permite compilar módulos en tiempo de ejecución (run time system) al código objeto generado.
  - Normalmente, los compiladores incrementales se utilizan en sistemas interactivos donde conviven módulos compilados con módulos modificables.

# TIPOS DE INTÉRPRETES

## Evaluadores Parciales

- La utilización de evaluadores parciales o especializados surge al considerar que muchos programas contienen dos tipos de datos de entrada.
  - Existen una serie de datos de entrada que son diferentes en cada ejecución mientras que otros datos no varían de una ejecución a otra.
  - El primer conjunto, se conoce como datos de entrada dinámicos (se denotará como  $Din$ ), mientras que el segundo conjunto, serían los datos de entrada estáticos ( $Est$ ).
- 
- Dado un programa  $P$ , el proceso de evaluación parcial consiste en construir otro programa especializado  $P_{Est}$  para los datos estáticos de  $P$ .
  - El programa  $P_{Est}$  suele estar escrito en el mismo lenguaje fuente que  $P$  y se debe garantizar que cuando se le presenten los datos dinámicos produzca los mismos resultados que si se hubiesen presentado todos los datos al programa  $P$  original.

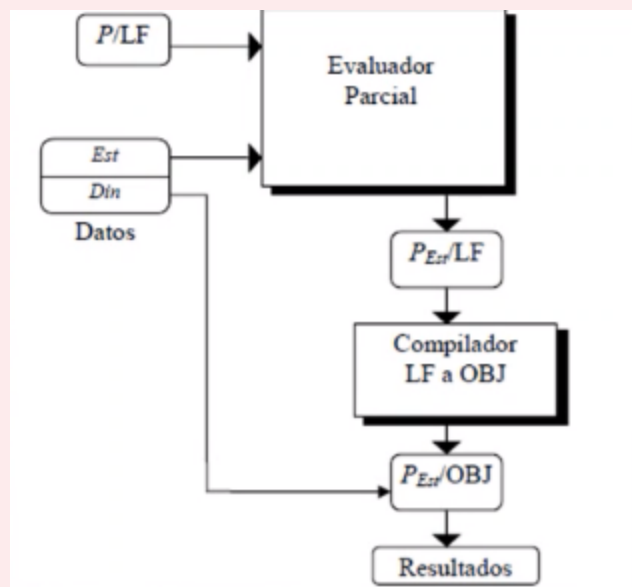


Figura 2. Evaluación Parcial

**Ejemplo:** Considérese el siguiente fragmento de programa P1 que toma la entrada de dos archivos diferentes, *fichEst* y *fichDin* y escribe el resultado en la salida estándar.

```
read(fichEst,a);
while (a > 0) do
begin
 read(fichDin,b);
 if (a > 10) then write (b - 2 * a)
 else write (a * a + b);
 read(fichEst,a);
end
```

Si el contenido de *fichEst* fuese siempre 5 12 7 -1, el evaluador parcial podría generar un programa especializado para dicho conjunto de datos, obteniendo  $P1_{Est}$ :

```
read(fichDin,b); write(25 + b);
read(fichDin,b); write(b - 24);
read(fichDin,b); write(49 + b);
```

La principal ventaja de la evaluación parcial es la eficiencia, si se conoce de antemano que un programa P va a ejecutarse muchas veces con un mismo conjunto de datos Est pero diferentes datos Din, será más eficiente evaluar parcialmente P para obtener  $P_{Est}$  y ejecutar luego  $P_{Est}$ .

En el ejemplo, puede observarse que es posible eliminar el bucle "while" y la sentencia "if" debido a que las condiciones dependen de datos estáticos, sin embargo, las cosas no son siempre tan fáciles, considérese que en el programa P1 se elimina la última sentencia "*read(fichEst,a)*" del bucle.

Entonces el evaluador parcial, podría entrar en un bucle infinito intentando generar el programa especializado.

Por este motivo, los evaluadores parciales deben realizar un complejo análisis del programa fuente para detectar que el proceso no genere un bucle infinito.

El *análisis de tiempo de enlace* (bindingtime analysis) es una técnica que se encarga de detectar qué valores son estáticos y pueden evaluarse y cuáles no.

Una aplicación interesante de la evaluación parcial es la posibilidad de generar compiladores a partir de intérpretes.

Para ello, supóngase que a la entrada del evaluador parcial se presenta el intérprete de un lenguaje de programación LF escrito en un lenguaje de transición LT, junto con un programa P escrito en LF.

El evaluador parcial generará un intérprete especializado para el programa P en el lenguaje LT.

Suponiendo la existencia de un compilador para el lenguaje LT, se puede obtener el intérprete especializado en código objeto que, una vez presentados los datos de entrada D genere los resultados.



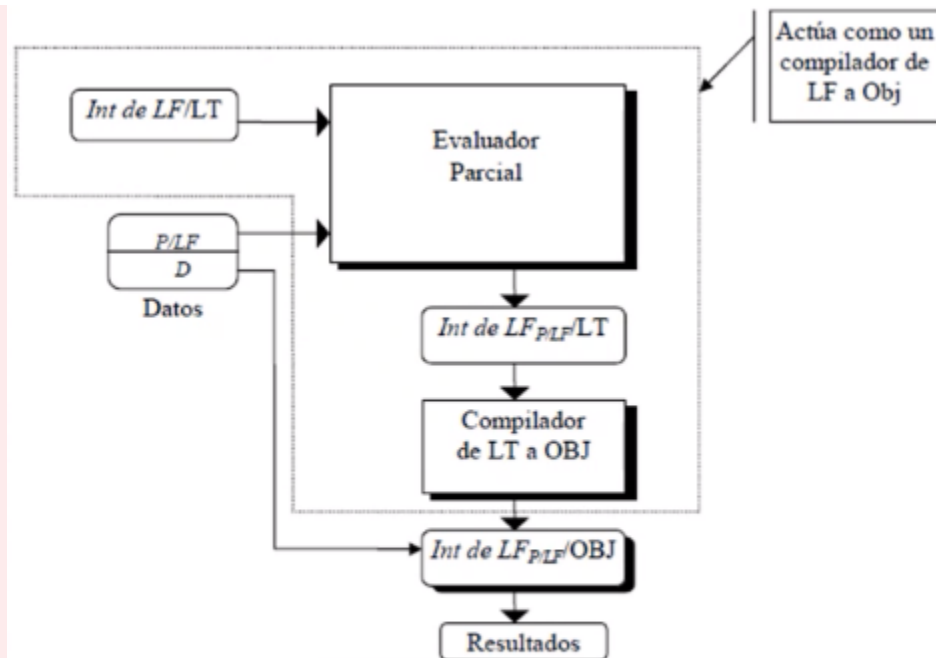


Figura 3. Obtención de un compilador a partir de un intérprete mediante evaluación parcial

Dado que los intérpretes se utilizan como especificaciones semánticas de un lenguaje, la obtención automatizada de un compilador a partir de la definición del intérprete permite alcanzar la eficiencia de un compilador sin perder la corrección semántica de un intérprete.

La evaluación parcial tiene otras aplicaciones interesantes en campos como el ray-tracing, modelización de mundos virtuales, reconocimiento de patrones, consultas a bases de datos, redes neuronales, etc.

▼ 21-09-2021

Lectura de la presentación Intérpretes 3

## TIPOS DE INTERPRETES

### Compiladores “Just in Time”

- Con la aparición de Internet surge la necesidad de distribuir programas de una forma independiente de la máquina permitiendo su ejecución en una amplia variedad de plataformas.
- Los códigos de bytes de la máquina Virtual de Java permiten la ejecución de programas distribuidos, ya que la mayoría de los visualizadores tienen un mecanismo capaz de interpretarlos.
- La interpretación de códigos de bytes supone una demora en los tiempos de ejecución.

Para evitar la interpretación, muchos sistemas transforman los códigos de bytes en código nativo siguiendo el modelo “Just in time”.

En este modelo, una unidad de compilación o clase se transmite en el formato de códigos de bytes, pero no se realiza la interpretación.

En lugar de ello, el código es compilado a código nativo justo en el momento en que lo necesita el programa que se está ejecutando.

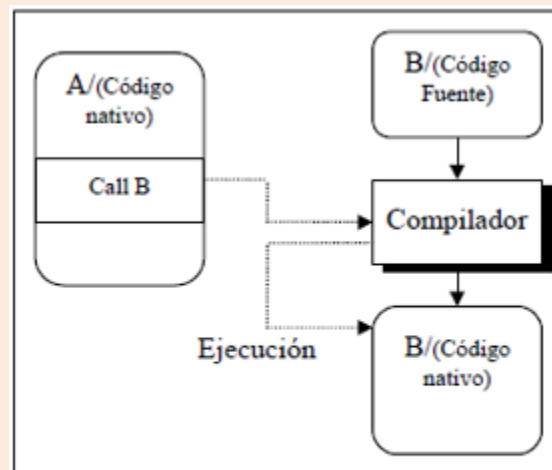


Figura 1. Compilación de una clase «Just in Time»

En la figura 1 se muestra el ejemplo de una unidad de compilación A compilada (en código nativo) que encuentra una instrucción de llamada a otra unidad B en código fuente (códigos de byte1).

El sistema realiza dos acciones:

Compila la unidad B a código nativo.  
Continúa la ejecución con el código nativo compilado de la unidad B

Las principales ventajas de la compilación Just in Time son:

1. Los programas grandes contienen porciones de código que no son ejecutadas en una ejecución típica del programa.

Por ejemplo, un visualizador de páginas Web, podría contener rutinas para manejar diversos formatos de los datos transmitidos, pero algunos de dichos formatos podrían no ser utilizados en una determinada ejecución del visualizador.

Puesto que la compilación Just in Time sólo traduce aquellas porciones de código que se necesitan, se evita el tener que compilar código que no se va a utilizar.

2. Los sistemas tradicionales realizan la compilación de todo el código antes de la ejecución, lo que para el usuario puede presentar un lapso de tiempo substancial entre el momento en que todas las unidades de compilación han sido transmitidas y el momento en que la ejecución puede comenzar.

Esta técnica tiende a repartir el tiempo de compilación a lo largo de la ejecución del programa.

El efecto producido al interrumpir la ejecución para compilar una unidad es similar al producido por la recolección de basura.

#### Compilación Continua

- La compilación continua surge como un intento de mejorar la compilación "Just in Time".
- El sistema mezcla el proceso de compilación a código nativo con el proceso de interpretación.
- Para conseguirlo, el sistema dispone de dos módulos: un módulo de interpretación de los códigos de bytes y otro módulo de compilación de códigos de bytes a código nativo.
- La idea consiste en que ambos módulos actúen a la vez (lo ideal sería disponer de dos procesadores), de forma que el sistema no se detenga a compilar un módulo, sino que vaya interpretándolo hasta que el compilador haya generado el código nativo.

En la figura 2 se distinguen los módulos que intervienen:

**Código:** El código contiene una mezcla de código fuente y código nativo del programa.

Inicialmente todo el código está sin compilar, pero a medida que el programa es ejecutado, el compilador genera traducciones a código nativo de las unidades de compilación.

**Compilador:** El módulo compilador traduce las unidades de compilación a código nativo.

A medida que se finaliza la traducción de una unidad, la versión en código nativo se deja disponible al intérprete.

**Intérprete:** El módulo intérprete se responsabiliza de la ejecución actual del programa.

Comienza interpretando el código fuente, haciendo saltos a las versiones en código nativo a medida que éstas están disponibles.

**Monitor:** Se encarga de coordinar la comunicación entre los dos módulos anteriores.

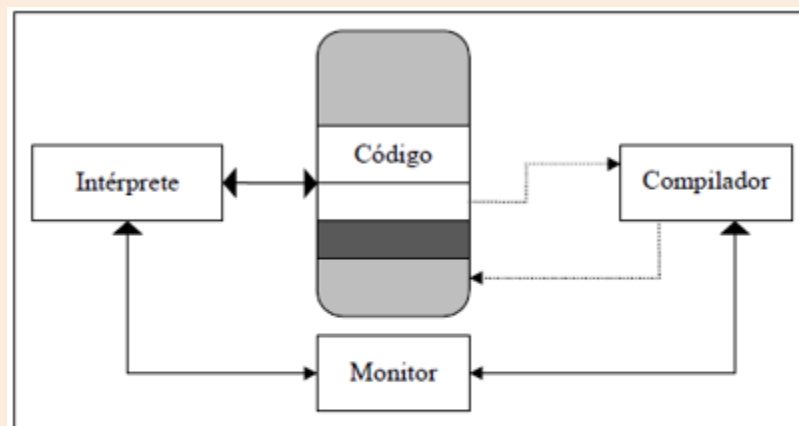


Figura 2. Compilación continua

La principal ventaja de la compilación continua respecto a la compilación Just in Time radica en no tener que esperar a compilar una unidad para comenzar su ejecución.

Plezbert estudian diferentes estrategias y se presenta con más detalle este modelo de compilación.



A modo de repaso de las estrategias tradicional, Just in time y continua, considérese el siguiente problema:

- “Desde una máquina cliente, desea ejecutarse un programa P cargado en un servidor. El programa consta de 3 módulos 'A', 'B' y 'C' en códigos de bytes. El tiempo de transmisión de cada módulo desde el servidor al cliente es de 2sg. El tiempo de compilación de códigos de bytes a código nativo es de 0.2sg por módulo y el tiempo de ejecución interpretando códigos de bytes es el doble del tiempo de ejecución en código nativo.
- Estudiar una ejecución particular que comienza por el módulo A, llama al módulo C y, tras ejecutar el módulo C, finaliza. Si dicha ejecución se realiza en código nativo, el módulo A tarda 1.4sg. mientras que el módulo C tarda 1sg. Los tiempos de transmisión de órdenes entre el servidor y el cliente se consideran despreciables.

Se trazará un diagrama de secuencias a fin de mostrar las diferencias entre cada estrategia.

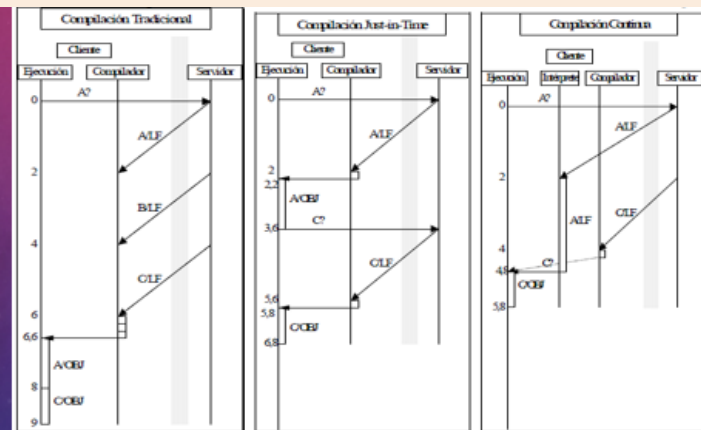


Figura 3. Diagrama de Secuencias comparando técnicas de compilación

En el esquema tradicional, el sistema solicita el módulo A al servidor y éste envía todos los módulos del programa (A, B y C).

Puesto que cada módulo tarda 2sg en transmitirse, la transmisión finaliza a los 6 sg, momento en el que se compilan los tres módulos (tardando 0,2 sg por módulo).

Al finalizar la compilación de los tres módulos, comienza la ejecución por el módulo A.

Por tanto, el tiempo de retardo, desde que el usuario solicita el módulo hasta que éste comienza a ejecutarse es de 6,6sg.

Tras ejecutarse el módulo A, se ejecuta el módulo C y el tiempo total de ejecución del programa serían 9 sg.

En el esquema Just-in-time el servidor envía solamente el módulo A (2 sg) , el cual es compilado en 0,2sg.

El tiempo de retardo será 2,2 sg. Al finalizar la ejecución de A, se solicita el módulo C al servidor, tras recibirlo y compilarlo, comienza su ejecución y el tiempo total de ejecución serán 6,8 sg.

Finalmente, en el esquema de compilación continua, tras solicitar y recibir el módulo A, el cliente comienza la ejecución interpretando códigos de bytes.

El tiempo de retardo para el usuario serán 2 sg .

El tiempo de ejecución del módulo A será 2,8 sg, el doble de 1,4 sg, puesto que se interpretan códigos de bytes.

Mientras se realiza la interpretación, el cliente solicita otros módulos.

En un caso óptimo, el cliente solicitaría el módulo C y éste se compilaría.

Al finalizar la ejecución de A, podría continuarse con la ejecución de C.

El tiempo de ejecución total sería 5,8 sg. Obsérvese que se ha considerado una situación óptima en la que el cliente solicite el módulo adecuado.

El cliente podría haberse equivocado y solicitar el módulo B, comportándose peor este esquema.

#### ▼ 22-09-2021

Actividad de investigación de un tipo de intérprete

#### ▼ 23-09-2021

Lectura de PDF sobre la elaboración de un intérprete

#### ▼ 24-09-2021

No hubo clase

#### ▼ 27-09-2021

Continuación del intérprete

#### ▼ 28-09-2021

Prolongación de la entrega del intérprete al domingo

#### ▼ 29-09-2021

No hubo clase

▼ **30-09-2021**

Revisión del intérprete

▼ **01-10-2021**

No hubo clase

▼ **04-10-2021**

No hubo clase

▼ **05-10-2021**

Exposiciones de intérpretes realmente no hubo clase

▼ **06-10-2021**

No hubo clase por el Congreso de Ciencias Exactas

▼ **07-10-2021**

No hubo clase por el Congreso de Ciencias Exactas

▼ **08-10-2021**

No hubo clase por el Congreso de Ciencias Exactas

▼ **11-10-2021**

Exposición de intérprete de nuestro team y team de Marco

▼ **12-10-2021**

No hubo clase

▼ **13-10-2021**

Exposición del team de Diego

▼ **14-10-2021**

No hubo clase

▼ **15-10-2021**

No hubo clase

## ▼ 18-10-2021

### Análisis léxico (compiladores)

La fase de rastreo o análisis léxico de un compilador, tiene la tarea de leer el programa fuente como un archivo de caracteres y dividirlo en tokens (como palabras de un lenguaje natural)

En cada caso un token representa cierto patrón de caracteres que el analizador léxico reconoce o ajusta desde el inicio de los caracteres de entrada restantes

El analizador léxico realiza una coincidencia de patrones → Expresiones regulares

El analizador léxico debe funcionar de manera tan eficiente como sea posible

Se divide el estudio de las cuestiones del analizador léxico de la siguiente forma:

1. Se da una perspectiva del analizador de estructuras y conceptos involucrados
2. Expresiones regulares
3. Máquinas de estados finitos para reconocer patrones de cadenas
4. Autómatas finitos fuera de las expresiones regulares
5. Métodos para procesos de reconocimiento por autómatas finitos
6. Generador de analizador léxico utilizando Lex disponible para Unix

## Proceso del análisis léxico

Leer caracteres de código fuente y formarlo en unidades lógicas llamadas tokens

En esto se asemeja a la tarea de deletreo

Los tokens caen en diversas categorías (palabras reservadas, símbolos especiales, cadenas, etc.)

Un analizador léxico calcula al menos tantos atributos de un token como sean necesarios para permitir el procesamiento siguiente



El analizador léxico funciona bajo el control del analizador sintáctico

▼ 19-10-2021

## Expresiones regulares

Son patrones de cadenas de caracteres

Una expresión regular  $r$  se encuentra completamente definida mediante el conjunto de cadenas

El conjunto se denomina lenguaje generado por la expresión regular escrito como  $L(r)$

Lenguaje = conjunto de cadenas

Este lenguaje depende del conjunto de caracteres que tenga disponible

Se trabaja con un alfabeto para las expresiones regulares

Los patrones serán marcados como negritas

Pueden haber metacaracteres y no pueden ser legales en el alfabeto pero debe utilizarse una convención de ambos

Un ejemplo de metacaracter es  $\backslash n$  = salto de línea

**Definición:** Dado cualquier caracter  $a$  del alfabeto  $\Sigma$  indicamos que la expresión regular  $a$  corresponde al caracter  $a$  escribiendo  $L(a) = \{a\}$

Hay otros símbolos que necesitaremos en situaciones especiales

Usamos  $\epsilon$  para denotar a la cadena vacía y definimos el metasímbolo  $\epsilon$  (en negritas) estableciendo que  $L(\epsilon) = \{\epsilon\}$

Luego ocupamos la ausencia de cadenas, es decir el conjunto vacío.  $\{\}$

$\{\}$  no contiene ninguna cadena mientras que  $\{\epsilon\}$  contiene la cadena simple que no contiene ningún caracter

## Operaciones de expresiones regulares

1. Selección entre alternativas que indica mediante el caracter  $|$ :

Si  $t$  y  $s$  son expresiones regulares, entonces  $r|s$  es una expresión regular que concuerda con  $r$  o con  $s$

## 2. Concatenación indicada como "y"

La concatenación de r y s se escribe como rs

La concatenación se puede definir como  $L(rs) = L(r)L(s)$  y se puede extender a más de dos expresiones regulares

## 3. Repetición o cerradura indicada como \*

### ▼ 20-10-2021

Para la repetición o cerradura, corresponde a cualquier concatenación finita de cadenas, por ejemplo,  $a^*$  corresponde a  $\varepsilon$ , a, aa, aaa, ...

Se puede escribir de la siguiente forma

$$S^* = \bigcup_{n=0}^{\infty} S^n$$

Podemos definir también  $L(r^*) = L(r)^*$

En realidad, la repetición tiene la más alta, la concatenación sigue y el operador "o" tiene la más baja

### ▼ 21-10-2021

No hubo clase

### ▼ 22-10-2021

No hubo clase

### ▼ 25-10-2021

No hubo clase

### ▼ 26-10-2021

# Compiladores de una sola pasada

Nombres para expresiones regulares:

A menudo es útil escribir un nombre para una expresión regular larga.

Como puede ser

$(1|2|3|4)$

digito\*digito

La idea de una expresión regular es escribir su nombre en cursiva o negrita y evitar las formas recursivas al momento de definir las

Una expresión regular es una de las siguientes:

1. Una expresión regular básica constituida por un sólo carácter **a**, donde a proviene de un alfabeto sigma de caracteres legales; el metacaracter épsilon o el metacaracter phi
2. Una expresión de la forma  $r|s$  donde r y s son expresiones regulares. En este caso  $L(r|s) = L(r) \cup L(s)$
3. Una expresión de la forma  $rs$  donde r y s son expresiones regulares. En este caso,  $L(rs) = L(r)L(s)$
4. Una expresión de la forma  $r^*$ , donde r es una expresión regular. En este caso  $L(r^*) = L(r)^*$
5. Una expresión regular de la forma  $(r)$  donde r es una expresión regular. En este caso  $L((r)) = L(r)$

La precedencia de las operaciones en 2, 3 y 4 está en orden inverso de su enumeración

También esta definición proporciona un significado de metacaracter a los seis símbolos phi, épsilon,  $|$ ,  $*$ ,  $($ ,  $)$ .

Ejemplo 1:

$(a|c)^*b(a|b)^*$

Ejemplo 2:

$(a|c)^*|(a|c)^*b(a|c)^*$

Ejemplo 3:

El conjunto:

$$S = \{b, aba, aabaa, aaabaaa, \dots\} = \{a^nba^n | n \neq 0\}$$

Este conjunto no se puede describir mediante una expresión regular, debido a que sólo se tiene una operación \* para hacer repetición pero permite cualquier número de repeticiones

Para proporcionar una demostración matemática requeriríamos el teorema que es el lema de extracción

Ejemplo 4:

$$(b(a|c))^* \rightarrow (a|c|ba|bc)^*(b|c)$$

**Tarea:** Hacer una investigación de qué es un compilador de una sola pasada

#### ▼ 27-10-2021

No hubo clase

#### ▼ 28-10-2021

Realización del segundo parcial

#### ▼ Tercer Parcial

#### ▼ 29-10-2021

No hubo clase

#### ▼ 01-11-2021



# AUTÓMATAS FINITOS

Los autómatas finitos o máquinas de estados finitos, son una manera matemática para describir clases particulares de algoritmos (o "máquinas").

En particular, los autómatas finitos se pueden utilizar para describir el proceso de reconocimiento de patrones en cadenas de entrada, y de este modo se pueden utilizar para construir analizadores léxicos.

Por supuesto, también existe una fuerte relación entre los autómatas finitos y las expresiones regulares.

Veremos como construir autómata finito a partir de una expresión regular.

Antes de comenzar, consideremos un ejemplo explicativo.

# AUTÓMATAS FINITOS

El patrón para identificadores como se define comúnmente en los lenguajes de programación está dado por la siguiente definición regular (supondremos que **letra** y **dígito** ya se definieron):

$$\text{identificador} = \text{letra}(\text{letra}|\text{dígito})^*$$

Esto representa una cadena que comienza con una letra y continúa con cualquier secuencia de letras y/o dígitos.

El proceso de reconocer una cadena así se puede describir mediante el diagrama de la Figura 1.

# AUTÓMATAS FINITOS

En este diagrama los círculos numerados 1 y 2 representan estados, que son localidades en el proceso de reconocimiento que registran cuánto del patrón ya se ha visto.

Las líneas con flechas representan transiciones que registran un cambio de un estado a otro en una coincidencia del carácter o caracteres mediante los cuales son etiquetados.

En el diagrama de muestra, el estado 1 es el estado de inicio, o el estado en el que comienza el proceso de reconocimiento.

Por convención, el estado de inicio se indica dibujando una línea con flechas sin etiqueta que proviene de "de ninguna parte".

El estado 2 representa el punto en el cual se ha igualado una sola letra (lo que se indica mediante la transición del estado 1 al estado 2 etiquetada con letra).

# AUTÓMATAS FINITOS

Una vez en el estado 2, cualquier número de letras y/o dígitos se puede ver, y una coincidencia de éstos nos regresa al estado 2.

Los estados que representan el fin del proceso de reconocimiento, en los cuales podemos declarar un éxito, se denominan estados de aceptación, y se indican dibujando un borde con línea doble alrededor del estado en el diagrama.

Puede haber más de uno de éstos.

En el diagrama de muestra el estado 2 es un estado de aceptación, lo cual indica que, después que cede una letra, cualquier secuencia de letras y dígitos subsiguiente (incluyendo la ausencia de todas) representa un identificador legal.

# AUTÓMATAS FINITOS

El proceso de reconocimiento de una cadena de caracteres real como un identificador ahora se puede indicar al enumerar la secuencia de estados y transiciones en el diagrama que se utiliza en el proceso de reconocimiento.

Por ejemplo, el proceso de reconocer **xtemp** como un identificador se puede indicar como sigue:

$$\rightarrow 1 \xrightarrow{x} 2 \xrightarrow{t} 2 \xrightarrow{e} 2 \xrightarrow{m} 2 \xrightarrow{p} 2$$

En este diagrama etiquetamos cada transición mediante la letra que se iguala en cada paso.

## DEFINICIÓN DE AUTÓMATAS FINITOS DETERMINÍSTICOS

Los diagramas como el que analizamos son descripciones útiles de los autómatas finitos porque nos permiten visualizar fácilmente las acciones del algoritmo.

Sin embargo, en ocasiones es necesario tener una descripción más formal de un autómata finito, y por ello procederemos ahora a proporcionar una definición matemática.

La mayor parte del tiempo, no obstante, no necesitaremos una visión tan abstracta como ésta, y describiremos la mayoría de los ejemplos en términos del diagrama solo.

Otras descripciones de autómatas finitos también son posibles, particularmente las tablas, y éstas serán útiles para convertir los algoritmos en código de trabajo.

Las describiremos a medida que surja la necesidad de utilizarlas.

También deberíamos advertir que lo que hemos estado describiendo son autómatas finitos **determinísticos**: autómatas donde el estado siguiente está dando unívocamente por el estado actual y el carácter de entrada actual.

Una generalización útil de esto es el **autómata finito no determinístico**, el cual se estudiará mas adelante.

Un AFD  $M$  se compone de un alfabeto  $\Sigma$ , un conjunto de estados  $S$ , una función de transición  $T: S * \Sigma \rightarrow S$ , un estado de inicio  $s_0 \in S$  y un conjunto de estados de aceptación  $A \subset S$ .

El lenguaje aceptado por  $M$ , escrito como  $L(M)$ , se define como el conjunto de cadenas de caracteres  $c_1, c_2, \dots, c_n$  con cada  $c_i \in \Sigma$ , tal que existen estados  $s_1 = T(s_0, c_1), s_2 = T(s_1, c_2), \dots, s_n = T(s_{n-1}, c_n)$ , con  $s_n$  como un elemento de  $A$  (es decir, un estado de aceptación).

Hacemos las anotaciones siguientes a esta definición  $S \times \Sigma$  se refiere al producto cartesiano o producto cruz de  $S$  y  $\Sigma$ : el conjunto de pares  $(s, c)$ , donde  $s \in S$  y  $c \in \Sigma$ .

La función  $T$  registra las transacciones:  $T(s, c) = s'$  si existe una transición del estado  $s$  al estado  $s'$  etiquetado mediante  $c$ .

El segmento correspondiente del diagrama para  $M$  tendrá el aspecto siguiente.



La aceptación como la existencia de una secuencia de estados  $s_1 = T(s_0, c_1), s_2 = T(s_1, c_2), \dots, s_n = T(s_{n-1}, c_n)$ , con  $s_n$  siendo un estado de aceptación, significa entonces lo mismo que el diagrama:

$$\rightarrow s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} s_2 \rightarrow \dots \rightarrow s_{n-1} \xrightarrow{c_n} s_n$$

Advertimos un número de diferencias entre la definición de AFD y el diagrama del ejemplo identificador.

#### ▼ 02-11-2021

No hubo clase. Asueto

#### ▼ 03-11-2021

Realización de un autómata finito para validar que un texto es un número real

Identificar 3 elementos del lenguaje por medio de una expresión regular y de un autómata finito



#### ▼ 04-11-2021

No hubo clase

#### ▼ 05-11-2021

División de exposiciones por equipos

#### ▼ 08-11-2021

##### ▼ Exposición Alejandro Camarillo

Traducción dirigida por sintaxis

El principio de traducción dirigida por la sintaxis establece que el significado de una construcción está directamente relacionado con su estructura sintáctica, que a su vez está representada en su árbol de análisis sintáctico. En este proceso de traducción dirigida por la sintaxis se asocia información a una construcción del lenguaje proporcionando atributos a los símbolos de la gramática que representan la construcción.

Una definición dirigida por sintaxis es una gramática de contexto libre en la que cada símbolo gramatical (terminales y no terminales) tiene un conjunto de atributos asociados, dividido en dos subconjuntos llamados atributos sintetizados y atributos heredados de dicho símbolo gramatical.

En una definición dirigida por sintaxis, para cada producción gramatical  $A \rightarrow \alpha$  se asocia un conjunto de reglas semánticas de la forma  $b := f(c_1, c_2, \dots, c_k)$  donde  $f$  es una función y  $b$  es un atributo sintetizado de  $A$  o un atributo heredado de uno de los símbolos gramaticales de la parte derecha de la producción y  $c_1, c_2, \dots, c_k$  son atributos que pertenecen a los símbolos gramaticales de la producción. Se dice que  $b$  depende de  $c_1, c_2, \dots, c_k$ .

Un atributo puede representar cualquier cosa, un nombre, una cadena, un número, un tipo, una posición de memoria, su valor en un nodo se define mediante una regla semántica asociada a la producción usada en dicho nodo.

En una definición dirigida por sintaxis se asume que los terminales sólo tienen atributos sintetizados, ya que la definición no proporciona ninguna regla semántica para los terminales. El analizador léxico es el que proporciona generalmente los valores para los atributos de los terminales.

Se puede decir que un atributo es sintetizado si su valor en un nodo del árbol de análisis sintáctico se determina a partir de los valores de atributos de los hijos de ese nodo (como decir de abajo hacia arriba). Se pueden calcular mediante un solo recorrido ascendente del árbol de análisis sintáctico, lo que es muy deseable.

Un atributo heredado es uno cuyo valor en un nodo de un árbol de análisis sintáctico está definido a partir de los atributos en el padre y los hermanos de dicho nodo. Éstos sirven para expresar la dependencia de una construcción de un lenguaje de programación en el contexto en el que aparece.

Si un atributo *b* en un nodo de un árbol de análisis sintáctico depende de un atributo *c*, entonces se debe evaluar la regla semántica para *b* en ese nodo después de la regla semántica que define a *c*. Las interdependencias entre los atributos heredados y sintetizados en los nodos de un árbol de análisis sintáctico se pueden representar de manera muy natural mediante un grafo dirigido llamado grafo de dependencias.

En el momento de la compilación, si el grafo de dependencias obtenido a partir del árbol de análisis sintáctico no contiene un ciclo, se obtiene un ordenamiento para cada entrada.

En el momento de la construcción del compilador, las reglas semánticas asociadas con las producciones son analizadas. Para cada producción el orden de evaluación de los atributos asociados queda predeterminado.

Escoge un orden sin considerar las reglas semánticas. Limita la clase de definiciones dirigidas por sintaxis que pueden implantarse.

Los métodos “sin recuerdo” y basados en reglas no necesitan construir de manera explícita el grafo de dependencias en el momento de la compilación, así que pueden ser más eficaces en el uso del tiempo y espacio del compilador.

El uso de árboles de análisis sintáctico como representación intermedia permite que la traducción se separe del análisis sintáctico. Las rutinas de traducción invocadas durante el análisis sintáctico deben activarse con dos clases de limitaciones. La primera, una gramática que resulte adecuada para el análisis sintáctico puede no reflejar la estructura jerárquica natural de las construcciones del lenguaje. La segunda, el método de análisis sintáctico restringe el orden en que se consideran los nodos de un árbol de análisis sintáctico. Pues este orden puede no coincidir con el orden en que se va disponiendo de la información sobre una construcción.

# ÁRBOLES SINTÁCTICOS

Un árbol sintáctico es una forma condensada de un árbol de análisis sintáctico, útil para representar construcciones de lenguajes. Los operadores y las palabras no aparecen como hojas sino más bien están asociadas con el nodo interior que sería el padre de dichas hojas en el árbol análisis sintáctico.

La construcción de un árbol sintáctico para una expresión es similar a la traducción de la expresión a una forma posfija. Se construyen subárboles para las subexpresiones creando un nodo para cada operador y para cada operando. Los hijos de un nodo de un operador son las raíces de los nodos que representan las subexpresiones que constituyen los operandos de dicho operador.

El uso de árboles de análisis sintáctico como representación intermedia permite que la traducción se separe del análisis sintáctico. Las rutinas de traducción invocadas durante el análisis sintáctico deben activarse con dos clases de limitaciones. La primera, una gramática que resulte adecuada para el análisis sintáctico puede no reflejar la estructura jerárquica natural de las construcciones del lenguaje. La segunda, el método de análisis sintáctico restringe el orden en que se consideran los nodos de un árbol de análisis sintáctico. Pues este orden puede no coincidir con el orden en que se va disponiendo de la información sobre una construcción.

## **GRAFOS DIRIGIDOS ACÍCLICOS PARA EXPRESIONES**

Sirve para identificar expresiones comunes. Tiene un nodo para cada subexpresión de la expresión, un nodo interior representa un operador y sus hijos representan los operandos. La única diferencia es que en un grafo dirigido acíclico que representa una subexpresión que aparece más de una vez, tendrá una cantidad de padres equivalente a la cantidad de veces que aparece.

Los atributos se pueden evaluar con un analizador sintáctico ascendente conforme la entrada es analizada. El analizador sintáctico puede conservar en su pila los valores de los atributos sintetizados asociados con los símbolos gramaticales. Siempre que se haga una reducción se calculan los valores de los nuevos atributos sintetizados a partir de los atributos que están en la pila para los símbolos gramaticales del lado derecho de la producción con la que se reduce.

# DEFINICIONES CON ATRIBUTOS POR LA IZQUIERDA

Cuando la traducción tiene lugar durante el análisis sintáctico, el orden de evaluación de los atributos va unido al orden en que, por el método de análisis sintáctico, se crean los nodos del árbol de análisis sintáctico.

Se les llama “por la izquierda” porque la información de los atributos parece fluir de izquierda a derecha. Las definiciones con atributos por la izquierda incluyen todas las definiciones dirigidas por sintaxis basadas en gramáticas LL(1). Toda definición con atributos sintetizados es una definición con atributos por la izquierda.

Un esquema de traducción es una gramática independiente de contexto en la que se asocian atributos con los símbolos gramaticales y se insertan acciones semánticas encerradas entre llaves { } dentro de los lados derechos de las producciones. Los esquemas de traducción pueden tener tanto atributos sintetizados como heredados.

Cuando se diseña un esquema de traducción, se deben respetar algunas limitaciones para asegurarse de que el valor de un atributo esté disponible cuando una acción se refiera a él.

# TRADUCCIÓN DESCENDENTE

Se trabaja con esquemas de traducción en lugar de hacerlo con definiciones dirigidas por sintaxis, así que se puede ser explícito en cuanto al orden en que tienen que lugar las acciones y las evaluaciones de los atributos.



# ELIMINACIÓN DE LA RECURSIVIDAD IZQUIERDA DE UN ESQUEMA DE TRADUCCIÓN

Como la mayoría de los operadores aritméticos son asociativos por la izquierda, es natural utilizar gramáticas recursivas por la izquierda para las expresiones. La transformación se aplica a esquemas de traducción con atributos sintetizados.

Para el análisis sintáctico descendente, se supone que una acción se ejecuta en el mismo momento en que se expandiría un símbolo en la misma posición. Un atributo heredado de un símbolo debe ser calculado por una acción que aparezca antes que el símbolo, y un atributo sintetizado del no terminal de la izquierda se debe calcular después de que hayan sido calculados todos los atributos de los que depende.

# DISEÑO DE UN TRADUCTOR PREDICTIVO

Un analizador sintáctico predictivo es un tipo especial de analizador sintáctico descendente recursivo en el que el símbolo del preanálisis determina sin ambigüedad el procedimiento seleccionado para cada no terminal. La secuencia de procedimientos llamados en el procesamiento de la entrada define implícitamente un árbol de análisis sintáctico para la entrada.

# EVALUACIÓN ASCENDENTE A ATRIBUTOS HEREDADOS

Existe un método, con el cual se puede implementar cualquier definición de una sintaxis que se base en atributos L, es decir, heredados por la izquierda en gramáticas LL(1) y muchos (no todos) basados en LR(1). Este método consiste en primero tomar el esquema de traducción orientada a sintaxis que posee las acciones semánticas incrustadas y remover dichas acciones, luego se debe insertar en su lugar un no terminal distinto para cada una de las producciones, el cual funciona como marca. Finalmente se agrega una producción extra a partir del no terminal marcador, que derive a epsilon y la acción a ejecutar (la acción removida anteriormente) [1].

a) Esquema de traducción por orientado a sintaxis

$E \rightarrow T R$

$R \rightarrow + T \{ \text{printf}(' + ') \} R \mid - T \{ \text{printf}(' - ') \} R \mid \epsilon$

$T \rightarrow \text{num} \{ \text{printf}(\text{num.val}) \}$

b) Modificación utilizando a M y N como no terminales  
marcadores

$E \rightarrow T R$

$R \rightarrow + T M R \mid - T N R \mid \epsilon$

$T \rightarrow \text{num} \{ \text{printf}(\text{num.val}) \}$

$M \rightarrow \epsilon \{ \text{printf}(' + ') \}$

$N \rightarrow \epsilon \{ \text{printf}(' - ') \}$

# EVALUADORES RECURSIVOS

Existen funciones recursivas que permiten recorrer un árbol de construido en base a una definición dirigida por la sintaxis. Las funciones recorren cada uno de los hijos para un determinado nodo, en algún orden, no necesariamente de izquierda a derecha [1]. Mediante la producción de un nodo la función puede determinar cuales son sus hijos, y retornar un valor para un no terminal A que puede recibir como parámetro algún valor heredado. El valor de retorno es utilizado para calcular el valor del no terminal del nodo en el nivel inmediato superior del árbol. El código correspondiente a cada producción simula la asociación de las reglas semánticas con dicha producción [1], las reglas son aplicadas conforme se calculen los valores heredados que cada no terminal recibe como parámetro.

# ESPACIO PARA EL VALOR DE LOS ATRIBUTOS EN TIEMPO DE COMPILACIÓN

El espacio en número de registros que utilizaran los valores de los atributos se puede calcular en tiempo de compilación basándose en el árbol de análisis sintáctico. El tiempo de vida para un atributo inicia cuando es primeramente computado y termina cuando todos los valores que dependen de este son calculados [1], es decir cuando dicho valor es usado por última vez.

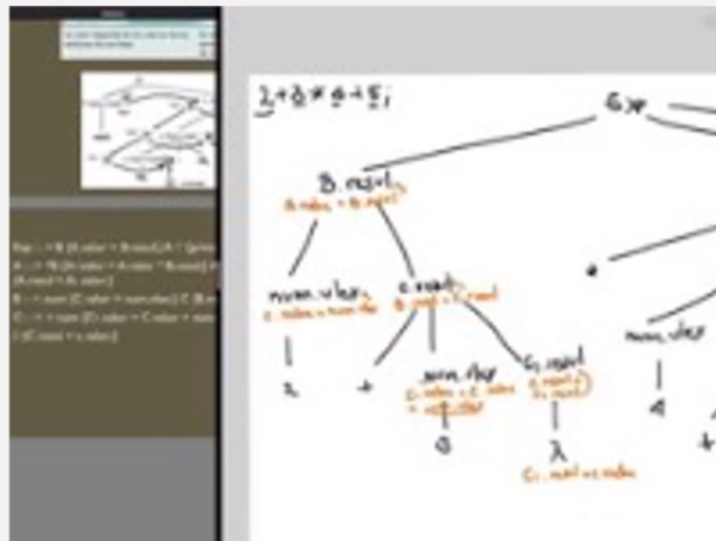
## ASIGNACIÓN DE ESPACIO EN TIEMPO DE CONSTRUCCIÓN - COMPILACIÓN

Se puede utilizar una pila para almacenar los valores de cada atributo, incluso no hay necesidad de preocuparse por las replicas innecesarias si los valores se guardan en la pila. Gracias a esta propiedad se puede predecir el tiempo de vida para los valores, sin tomar en cuenta que estos pueden estar compartidos con otras variables, lo que facilita la predicción.

# ANÁLISIS DE LAS DEFINICIONES DIRIGIDAS POR SINTAXIS

Las funciones para un no terminal mapen los valores de los atributos heredados de un nodo con atributos sintetizados. Mientras que para los terminales se debe realizar una función por aparte, es decir distinta de la de los no terminales, sin embargo los terminales se pueden considerar como un solo grupo y ser evaluados con una simple función [1], la agrupación de dichos atributos esta determinada por las dependencias que se presentan con el conjunto de reglas semánticas especificadas en la definición orientada a la sintaxis.

## EJEMPLO: TRADUCCIÓN DIRIGIDA POR LA SINTAXIS





# PREGUNTAS:

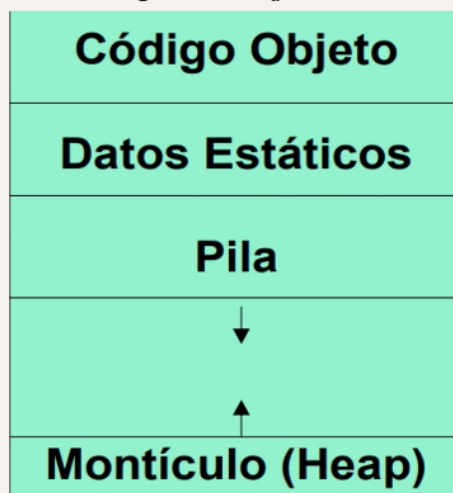
- ¿Qué son los atributos heredados?
- ¿En que consisten los grafos de dependencias?
- ¿Qué es una esquema de traducción?
- Menciona y describe 2 tipos de evaluacion de reglas semánticas
- ¿Qué son los atributos sintetizados?
- Menciona los esquemas de traducción

## ▼ 09-11-2021

Nuestra exposición (todo el team secuestro de verificación de tipos)

## ▼ 10-11-2021

### Organización de la Memoria en Tiempo de Ejecución



### Registros de Activación

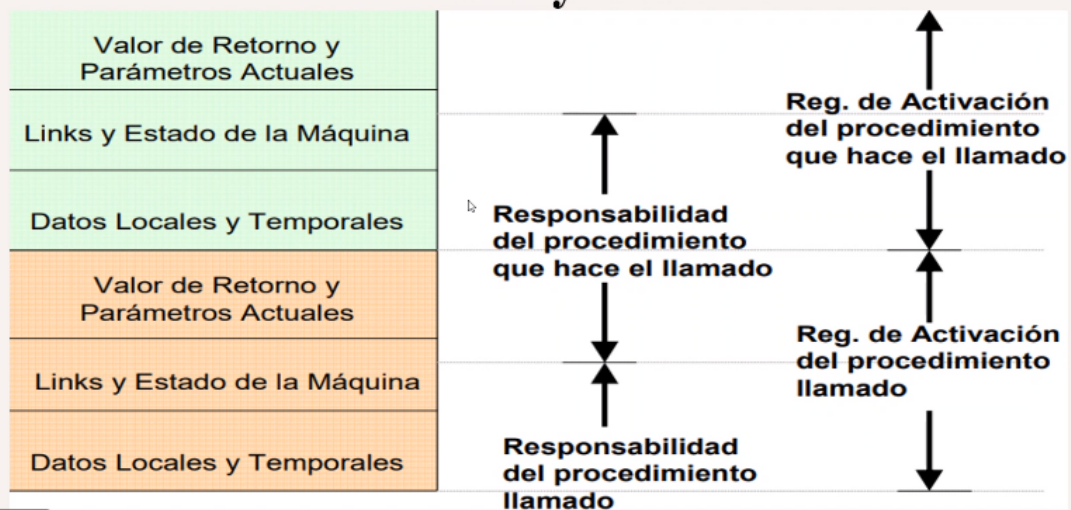
|                                      |
|--------------------------------------|
| <b>Valor Retorno</b>                 |
| <b>Parámetros Actuales</b>           |
| <b>Link de Control</b><br>(opcional) |
| <b>Link de Acceso</b><br>(opcional)  |
| <b>Estado de la Máquina</b>          |
| <b>Datos Locales</b>                 |
| <b>Temporales</b>                    |

## Asignación en Pila: Secuencias de Llamado y Retorno

- **Llamado:** Se genera en el código destino.
  - o Evalúa los parámetros actuales.
  - o Guarda la dirección de retorno y el valor del tope de la pila.
  - o Incrementa el apuntador del tope de la pila.
  - o Guarda el estado de la máquina.
  - o Inicializa los datos locales e inicia la ejecución.
- **Retorno:** Restaura el estado de la máquina de tal forma que el procedimiento llamador pueda continuar su ejecución.
  - o Ubica el valor de retorno.
  - o Restaura el tope de la pila y el estado de la máquina.
  - o Copia el valor de retorno a su propio registro de activación

Simular microtubo

## Asignación en Pila: Secuencias de Llamado y Retorno



## Estrategias para Asignación de Memoria: Asignación Estática



Los nombres se enlazan a las posiciones de memoria durante la compilación del programa.

No requiere de un programa run-time.

Los valores de las variables locales permanecen entre diferentes activaciones del mismo procedimiento

Esto implica que:

- ∞ El tamaño de los objetos de datos debe ser conocido en tiempo de compilación.
- ∞ No es posible declarar procedimientos recursivos.
- ∞ No pueden usarse estructuras de datos dinámicas.

**Ejemplo:** Fortran

## Estrategias para Asignación de Memoria: Asignación en Pila



**Implementa la pila de control:** El almacenamiento es manejado como una pila, y los registros de activación entran a la pila cuando inicia la ejecución del procedimiento.

Los valores de las variables locales se pierden cuando la activación termina.

Un registro marca el tope de la pila.

## Estrategias para Asignación de Memoria: Asignación en Heap



Usado para guardar datos no estáticos, cuyo tiempo de vida no está limitado por el tiempo de activación de un procedimiento.

Requiere un Administrador de Memoria (Memory Management) que localiza (allocation) y libera (deallocation) espacio en el heap, y un Recolector de Basura (Garbage Collection) que recupera espacios de memoria que ya no están siendo utilizados.

# Asignación en Heap: Asignación Dinámica de Memoria

Técnicas para implementarla:

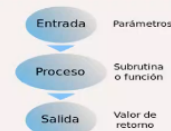
™ Con asignación de **memoria explícita**:

- Bloques de Tamaño Fijo.
- Bloques de Tamaño Variable.

™ Con asignación de **memoria implícita**:

- Requiere un paquete run-time.
- Los bloques de memoria tienen un formato definido (pueden ser de tamaño fijo o variable).
- El run-time debe reconocer si un bloque de memoria continua en uso.

## Métodos para Paso de Parámetros



- △ **Por valor:** El procedimiento llamador evalúa los parámetros y pasa su r-valor al procedimiento llamado.
- △ **Por referencia:** El procedimiento llamador evalúa los parámetros y pasa su l-valor al procedimiento llamado (si es una expresión le asigna memoria, y pasa la dirección de memoria asignada).
- △ **Copiar-Restaurar:** El procedimiento llamador evalúa los parámetros actuales, y pasa los r-valores al procedimiento llamado. Cuando el control retorna, los r-valores de los parámetros formales son copiados en los parámetros actuales.

**Ej.** Algunas implementaciones de Fortran.

- △ **Por nombre:** El procedimiento actúa como una macro, cuyo cuerpo se copia en el procedimiento llamador, los parámetros actuales sustituyen a los formales.

## Acceso a Nombres no Locales: Alcance Léxico

Determina que declaración aplica a un nombre, examinando el programa (el texto).

### Sin Procedimientos Anidados:

- Ω Todo nombre no local es global (estático).
- Ω Se implementa fácilmente el paso de procedimientos como parámetros, o el retorno de estos como resultado.

### Con Procedimientos Anidados:

- Ω Para un nombre no local aplica la declaración más cercana de él.
- Ω La anidación en la declaración no implica anidación en el llamado.
- Ω **Profundidad de anidamiento:** En un procedimiento asocia el nombre no local con la profundidad a la cual fue declarado.
- Ω El link de acceso apunta al registro de la activación mas reciente del procedimiento que lo anida.
- Ω Recorriendo los links de acceso puede llegar al registro de activación que tiene el nombre.



## Acceso a Nombres no Locales: Alcance Léxico

### Con Procedimientos Anidados:

Si  $p$  tiene profundidad  $n_p$  y  $q$  profundidad  $n_q$ :

- Si  $p$  anida en  $q$ ,  $n_p > n_q$
- $n_p = n_q$ , ssi  $p = q$
- Si  $p$  accede a una variable  $x$  de  $q$ ,  $n_p \geq n_q$ , se debe seguir el link de acceso ( $n_p - n_q$ ) veces (esta cantidad se conoce en tiempo de compilación).

Si  $n_p > n_q$ , entonces  $n_p = n_q + 1$ , el link de acceso de  $p$  apunta a  $q$ .

Si  $n_p = n_q$ ,  $p = q$ , es recursivo,  $q$  copia el link de  $p$ .

Si  $n_p < n_q$ , entonces  $q$  anida en un procedimiento  $r$ , debe saltar  $n_q - n_p + 1$  links de acceso para encontrar la activación del procedimiento  $r$ .

Si  $p$  es recibido por  $q$  como parámetro, pasar como información del parámetro el procedimiento y el link de acceso.

## Bibliografías

- Anónimo. (2021). *Compilador Diseño - Entorno de Ejecución en Tiempo Real*. Noviembre 07, 2021, de Tutorialspoint Sitio web: [https://www.tutorialspoint.com/es/compiler\\_design/compiler\\_design\\_runtime\\_environment.htm](https://www.tutorialspoint.com/es/compiler_design/compiler_design_runtime_environment.htm)
- Acevedo M. (2018). *LENGUAJE Y COMPILADORES*. Noviembre 07, 2021, de ISSUU Sitio web: [https://issuu.com/manuelacevedo1986/docs/unidad\\_iv](https://issuu.com/manuelacevedo1986/docs/unidad_iv)
- Pabón C. (2019). *Compiladores: Ambientes para Ejecución*. Noviembre 07, 2021, de Universidad Javeriana Cali Sitio web: [http://cic.javerianacali.edu.co/wiki/lib/exe/fetch.php?media=materias:compilers:compilers\\_23-24-25\\_2008\\_1.pdf](http://cic.javerianacali.edu.co/wiki/lib/exe/fetch.php?media=materias:compilers:compilers_23-24-25_2008_1.pdf)

### ▼ 11-11-2021

No hubo clase

### ▼ 12-11-2021

Exposición del equipo de Yissel

### ▼ 15-11-2021

No hubo clase. Asueto

### ▼ 16-11-2021

Exposición de Roberto sobre Partes del Compilador

!!! NO HE PASADO APUNTES DESDE AQUÍ !!!

# Autómatas Celulares

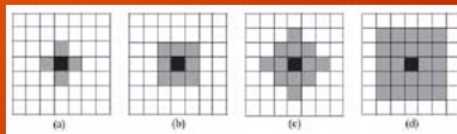
## Elementos de un AC

- Configuración Inicial.

Es la asignación inicial de un estado a cada una de las células del espacio.

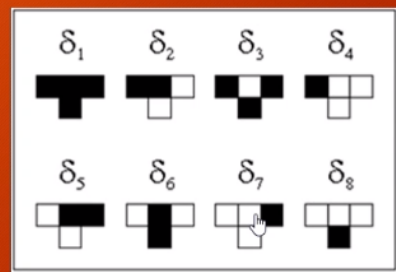
- Vecindades.

Define el conjunto de células que se consideran adyacentes a una dada, así como la posición relativa respecto a ella. Cuando el espacio es uniforme, la vecindad de cada célula es isomorfa (es decir, que tiene el mismo aspecto)



- Función de Transición Local.

Es la regla de evolución que determina el comportamiento del AC. Se calcula a partir del estado de la célula y su vecindad. Define cómo debe cambiar de estado cada célula dependiendo su estado anterior y de los estados anteriores de su vecindad. Suele darse como una expresión algebraica o un grupo de ecuaciones.



Adicionalmente, es necesario especificar el tipo de limite o frontera del espacio, entre los que podemos destacar:

- **Frontera Abierta.**

Se considera que todas las células fuera del espacio del autómatas toman un valor fijo.

- **Frontera Reflectora.**

Las células fuera del espacio del autómatas toman los valores que están dentro, como si se tratara de un espejo.

Es importante destacar la complejidad que se logra con un AC.

De acuerdo a la dimensión en la que se genere (línea, plano, espacio, etc.) tendrá un numero potencial de vecinos.

Por ejemplo, supongamos que la vecindad se considera de radio 1, es decir, solo las células inmediatas o mas cercanas serán tomadas en cuenta.

Para el caso de una sola dimensión cada célula tendrá solo 2 vecinos.

Para un AC en dos dimensiones contara con 4 (arriba, abajo, izquierda, derecha) u 8 vecinas si tomamos en cuenta también las diagonales.

Y en el caso de un AC en 3D llegara a tener hasta 26 vecinos cada célula.

Como el siguiente estado de cada célula se computa en base al estado anterior y de sus vecinas (es decir que el estado 2 surge del estado 1 y el 3 del 2,etc.), se puede tener un sistema con una función de transición muy rica y que, a veces, puede presentar un comportamiento muy difícil de predecir.



Stephen Wolfram comenzó a trabajar en autómatas celulares a mediados de 1981 después de considerar cómo los patrones complejos parecían formarse en la naturaleza en violación de la segunda ley de la termodinámica.

Sus investigaciones fueron inicialmente impulsados por un interés en sistemas de modelado, como las redes neuronales.

Tras ver la inesperada complejidad del comportamiento de estas reglas simples Wolfram llevó a sospechar que la complejidad en la naturaleza puede ser debida a mecanismos similares.

En 2002 Wolfram publicó un texto de 1.280 páginas, *A New Kind of Science*, que sostiene ampliamente que los descubrimientos sobre autómatas celulares no son hechos aislados sino que son robustos y tienen importancia para todas las disciplinas de la ciencia.

Wolfram define cuatro clases en las que los AC.

Mientras que los estudios anteriores en autómatas celulares tienden a tratar de identificar el tipo de patrones de reglas específicas, la clasificación de Wolfram fue el primer intento de clasificación global.

En orden de complejidad las clases que identifica son:

- Clase I: Casi todos los patrones iniciales evolucionan rápidamente en un estado estable y homogéneo. Cualquier aleatoriedad en el patrón inicial desaparece.
- Clase II: Casi todos los patrones iniciales evolucionan rápidamente hacia estructuras estables u oscilantes. Parte de la aleatoriedad del patrón inicial puede permanecer, pero solo algunos restos. Los cambios locales en el patrón inicial tienden a permanecer locales.
- Clase III: Casi todos los patrones iniciales evolucionan de forma pseudo-aleatoria o caótica. Las estructuras estables que aparecen son destruidas rápidamente por el ruido circundante. Los cambios locales en el patrón inicial tienden a propagarse indefinidamente.



- Clase IV: Casi todos los patrones iniciales evolucionan en las estructuras que interactúan de manera compleja e interesante, con la formación de las estructuras locales que son capaces de sobrevivir por largos períodos de tiempo. Podría ser el caso de que apareciesen estructuras estables u oscilantes, pero el número de pasos necesarios para llegar a este estado puede ser muy grande, incluso cuando el patrón inicial es relativamente simple. Los cambios locales en el patrón inicial pueden extenderse indefinidamente. Wolfram ha conjeturado que muchos, si no todos, los AC de esta clase son capaces de realizar computación universal. Algo que ha sido demostrado para el autómata 110 y para el juego de la vida de John Conway.

Estas definiciones son de carácter cualitativo y hay cierto margen de interpretación. Según Wolfram, "... con casi cualquier esquema de clasificación general, hay casos en los que, inevitablemente, se asignan a una clase por una definición y otra clase por otra definición, y lo mismo ocurre con los autómatas celulares: hay veces que las reglas muestran algunas de las características de una clase y otras veces de otra clase".

▼ 18-11-2021

No hubo clase

▼ 19-11-2021

No hubo clase

▼ 22-11-2021

Más trabajo

▼ 23-11-2021

No hubo clase

▼ 24-11-2021

▼ 25-11-2021

▼ 26-11-2021