



Instituto tecnológico de Iztapalapa

Ingeniería en Sistemas Computacionales

Lenguajes y automatas 2

Santana González Jesús Salvador: 171080127

Cabrera Ramírez Gerardo: 171080187

Morales Carrillo Gerardo: 171080120

Actividad semana 14

16/10/2020

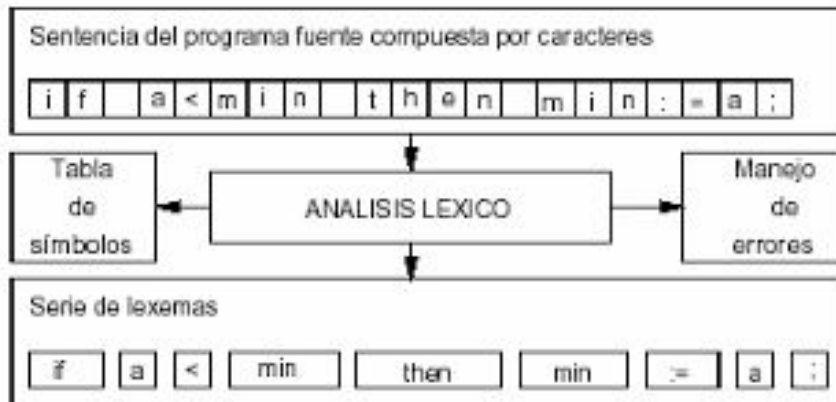


## "Análisis Léxico parte 1"

Conceptos:

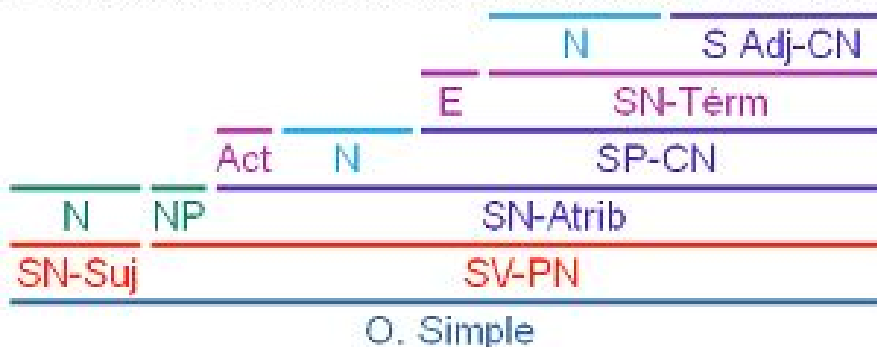
Análisis de léxico

A la primera fase de un compilador se le llama análisis de léxico o escaneo. El analizador de léxico lee el flujo de caracteres que componen el programa fuente y los agrupa en secuencias.

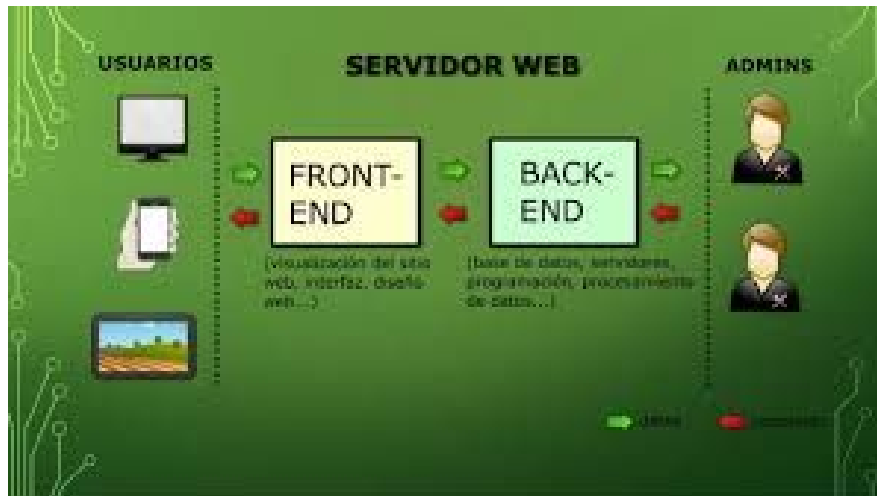


**Análisis sintáctico** La segunda fase del compilador es el análisis sintáctico o parsing. El parser (analizador sintáctico) utiliza los primeros componentes de los tokens producidos por el analizador de léxico para crear una representación intermedia en forma de árbol que describa la estructura gramatical del flujo de tokens.

EdAS es un editor de análisis sintáctico



**Front End:** El Front End es la parte del compilador que interactúa con el usuario y por lo general, es independiente de la plataforma en la que se trabaja.



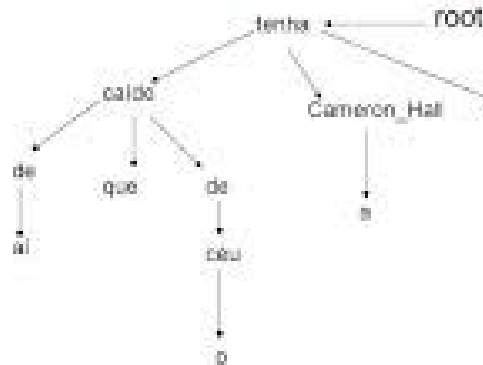
Back End: Esta parte del compilador es la encargada de generar el código en formato de máquina, a partir del trabajo hecho por el Front End.



Gráficos de dependencias Un grafo de dependencias describe el flujo de información entre las instancias de atributos en un árbol de análisis sintáctico específico; una flecha de una instancia de atributo a otra significa que el valor de la primera se necesita para calcular la segunda.

## Ejemplo de Grafo de dependencias (1)

1	De	7
2	al	1
3	que	7
4	a	5
5	Cameron_Hall	8
6	tenha	0
7	caldo	6
8	de	7
9	o	10
10	cóu	8
11		6



Unlabeled Parsing de Dependencias

Gestión de información de errores.

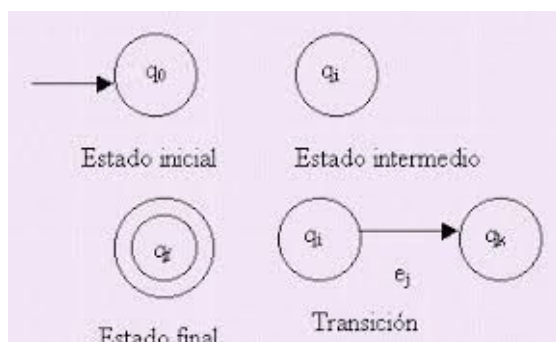
Si los compiladores tuvieran que procesar solamente programas correctos, su diseño e implementación se simplificará en buena medida. Pero los programadores escriben programas incorrectos frecuentemente, y un buen compilador debe ayudar al programador a localizar e identificar los errores.

Generación del código objeto.

La fase final de un compilador es la de generación del código objeto, consistente en código máquina o código ensamblador.

Representación de lenguajes:

En general existen dos esquemas diferentes para definir un Lenguaje, los cuales se conocen como esquema generador y esquema reconocedor. En el caso de los esquemas generadores se trata de un mecanismo que nos permite “generar” las diferentes sentencias del lenguaje





Concepto de gramática:

La gramática es un ente o modelo matemático que permite especificar un lenguaje, es decir, es el conjunto de reglas capaces de generar todas las posibilidades combinatorias de ese lenguaje, y sólo las de dicho lenguaje, ya sea éste un lenguaje formal o un lenguaje natural.

Santana Gonzales Jesús Salvador

"Análisis Léxico (partes 2 y 3)" y "Análisis Sintáctico (parte 1)"

### Funciones del analizador léxico y sus ventajas

El analizador léxico realiza varias funciones, siendo la fundamental la de agrupar los caracteres que va leyendo uno a uno del programa fuente y formar los tokens.

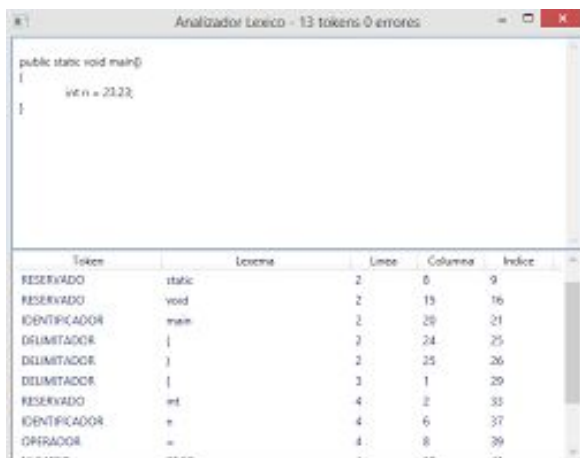
**Implementación de un analizador léxico** Hay varias formas de implementar un analizador léxico:

**Utilizando un generador de analizadores léxicos:** son herramientas que a partir de las expresiones regulares generan un programa que permite reconocer los tokens o componentes léxicos. Estos programas suelen estar escritos en C, donde una de las herramientas es FLEX, o pueden estar escritos en Java, donde las herramientas pueden ser JFLEX o JLEX.

**Utilizando un lenguaje de alto nivel:** a partir del diagrama de transiciones y del pseudocódigo correspondiente se programa el analizador léxico (véase un ejemplo en Loudon, 2004).

### Errores léxicos

Los errores léxicos son detectados, cuando durante el proceso de reconocimiento de caracteres, los símbolos que tenemos en la entrada no concuerdan con ningún patrón.



Token	Lexema	Linea	Columna	Indice
RESERVADO	static	2	6	9
RESERVADO	void	2	15	16
IDENTIFICADOR	main	2	20	21
DELIMITADOR	{	2	24	25
DELIMITADOR	}	2	25	26
DELIMITADOR		3	1	29
RESERVADO	int	4	2	33
IDENTIFICADOR	n	4	6	37
OPERADOR	=	4	8	39
DELIMITADOR	.	4	13	43



## ¿Qué es el analizador sintáctico ?

Es la fase del analizador que se encarga de chequear el texto de entrada en base a una gramática dada. Y en caso de que el programa de entrada sea válido, suministra el árbol sintáctico que lo reconoce. En teoría, se supone que la salida del analizador sintáctico es alguna representación del árbol sintáctico que reconoce la secuencia de tokens suministrada por el analizador léxico. En la práctica, el analizador sintáctico también hace:

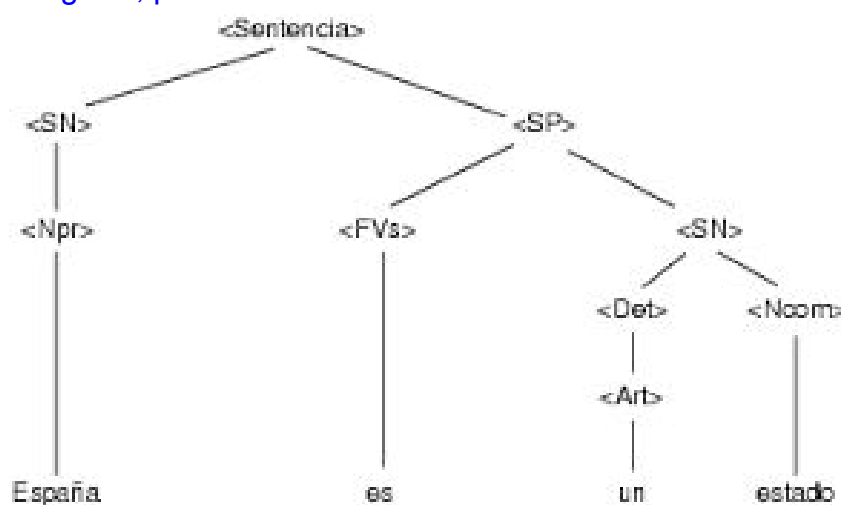
- Acceder a la tabla de símbolos (para hacer parte del trabajo del analizador semántico).
- Chequeo de tipos ( del analizador semántico).
- Generar código intermedio.
- Generar errores cuando se producen.

En definitiva, realiza casi todas las operaciones de la compilación. Este método de trabajo da lugar a los métodos de compilación dirigidos por sintaxis.

## Manejo de errores sintácticos

Si un compilador tuviera que procesar sólo programas correctos, su diseño e implantación se simplificarían mucho. Pero los programadores a menudo escriben programas incorrectos, y un buen compilador debería ayudar al programador a identificar y localizar errores. Es más, considerar desde el principio el manejo de errores puede simplificar la estructura de un compilador y mejorar su respuesta a los errores. Los errores en la programación pueden ser de los siguientes tipos:

- Léxicos, producidos al escribir mal un identificador, una palabra clave o un operador.
- Sintácticos, por una expresión aritmética o paréntesis no equilibrados.
- Semánticos, como un operador aplicado a un operando incompatible.
- Lógicos, puede ser una llamada infinitamente recursiva.



Santana Gonzalez Jesus Salvador



## ANÁLISIS SEMÁNTICO EN PROCESADORES DE LENGUAJE

La fase de análisis semántico de un procesador de lenguaje es aquella que computa la información adicional necesaria para el procesamiento de un lenguaje, una vez que la estructura sintáctica de un programa haya sido obtenida. Es por tanto la fase posterior a la de análisis sintáctico y la última dentro del proceso de síntesis de un lenguaje de programación .

Sintaxis de un lenguaje de programación es el conjunto de reglas formales que especifican la estructura de los programas pertenecientes a dicho lenguaje.

Semántica de un lenguaje de programación es el conjunto de reglas que especifican el significado de cualquier sentencia sintácticamente válida. Finalmente, el análisis semántico<sup>1</sup> de un procesador de lenguaje es la fase encargada de detectar la validez semántica de las sentencias aceptadas por el analizador sintáctico.

La sintaxis del lenguaje C indica que las expresiones se pueden formar con un conjunto de operadores y un conjunto de elementos básicos. Entre los operadores, con sintaxis binaria infija, se encuentran la asignación, el producto y la división. Entre los elementos básicos de una expresión existen los identificadores y las constantes enteras sin signo (entre otros).

Su semántica identifica que en el registro asociado al identificador superficie se le va a asociar el valor resultante del producto de los valores asociados a base y altura, divididos por dos (la superficie de un triángulo).

### Especificación Semántica de Lenguajes de Programación

Existen dos formas de describir la semántica de un lenguaje de programación: mediante especificación informal o natural y formal. La descripción informal de un lenguaje de programación es llevada a cabo mediante el lenguaje natural. Esto hace que la especificación sea inteligible (en principio) para cualquier persona.

La experiencia nos dice que es una tarea muy compleja, si no imposible, el describir todas las características de un lenguaje de programación de un modo preciso. Como caso particular, véase la especificación del lenguaje ISO/ANSI C++ [ANSIC++]. La descripción formal de la semántica de lenguajes de programación es la descripción rigurosa del significado o comportamiento de programas, lenguajes de programación, máquinas abstractas o incluso cualquier dispositivo hardware.

- Revelar posibles ambigüedades existentes implementaciones de procesadores de lenguajes o en documentos descriptivos de lenguajes de programación.
  - Ser utilizados como base para la implementación de procesadores de lenguaje.
  - Verificar propiedades de programas en relación con pruebas de corrección o información relacionada con su ejecución.



- Diseñar nuevos lenguajes de programación, permitiendo registrar decisiones sobre construcciones particulares del lenguaje, así como permitir descubrir posibles irregularidades u omisiones.
- Facilitar la comprensión de los lenguajes por parte del programador y como mecanismo de comunicación entre diseñador del lenguaje, implementador y programador. La especificación semántica de un lenguaje, como documento de referencia, aclara el comportamiento del lenguaje y sus diversas construcciones.
  - Estandarizar lenguajes mediante la publicación de su semántica de un modo no ambiguo. Los programas deben poder procesarse en otra implementación de procesador del mismo lenguaje exhibiendo el mismo comportamiento.

### **Especificación Formal de Semántica**

Si bien la especificación formal de la sintaxis de un lenguaje se suele llevar a cabo mediante la descripción estándar de su gramática en notación BNF (Backus-Naur Form), en el caso de la especificación semántica la situación no está tan clara; no hay ningún método estándar globalmente extendido. El comportamiento de las distintas construcciones de un lenguaje de programación, puede ser descrito desde distintos puntos de vista.

#### **Tareas y Objetivos del Análisis Semántico**

El análisis semántico<sup>11</sup> de un procesador de lenguaje es la fase encargada de detectar la validez semántica de las sentencias aceptadas por el analizador sintáctico.

### **Comprobaciones pospuestas por el analizador sintáctico**

A la hora de implementar un procesador de un lenguaje de programación, es común encontrarse con situaciones en las que una gramática libre de contexto puede representar sintácticamente propiedades del lenguaje; sin embargo, la gramática resultante es compleja y difícil de procesar en la fase de análisis sintáctico. En estos casos es común ver cómo el desarrollador del compilador escribe una gramática más sencilla que no representa detalles del lenguaje, aceptándolos como válidos cuando realmente no pertenecen al lenguaje.

### **Comprobaciones dinámicas**

Todas las comprobaciones semánticas descritas en este punto suelen llevarse a cabo en fase de compilación y por ello reciben el nombre de “estáticas”. Existen comprobaciones que, en su caso más general, sólo pueden ser llevadas a cabo en tiempo de ejecución y por ello se llaman “dinámicas”. Éstas suelen ser comprobadas por un intérprete o por código de comprobación generado por el compilador –también puede darse el caso de que no se comprueben. Diversos ejemplos



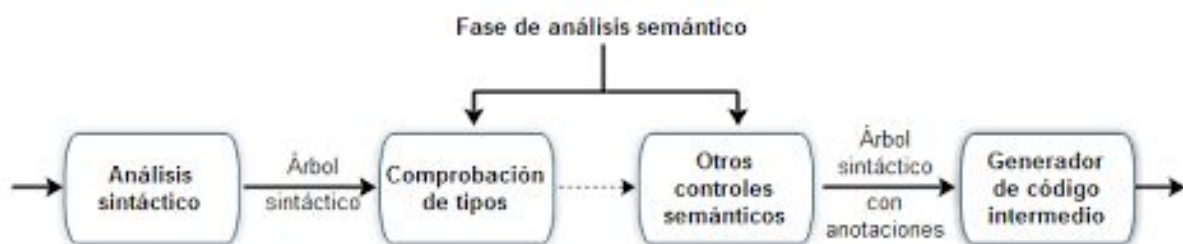


pueden ser acceso a un vector fuera de rango, utilización de un puntero nulo o división por cero.

### Comprobaciones de tipo

Sin duda, este tipo de comprobaciones es el más exhaustivo y amplio en fase de análisis semántico. Ya bien sea de un modo estático (en tiempo de compilación), dinámico (en tiempo de ejecución) o en ambos, las comprobaciones de tipo son necesarias en todo lenguaje de alto nivel. De un modo somero, el analizador semántico deberá llevar a cabo las dos siguientes tareas relacionadas con los tipos: Tareas y Objetivos del Análisis Semántico.

Comprobar las operaciones que se pueden aplicar a cada construcción del lenguaje. Dado un elemento del lenguaje, su tipo identifica las operaciones que sobre él se pueden aplicar. Por ejemplo, en el lenguaje Java el operador de producto no es aplicable a una referencia a un objeto. De un modo contrario, el operador punto sí es válido. 2. Inferir el tipo de cada construcción del lenguaje. Para poder implementar la comprobación anterior, es necesario conocer el tipo de toda construcción sintácticamente válida del lenguaje. Así, el analizador semántico deberá aplicar las distintas reglas de inferencia de tipos descritas en la especificación del lenguaje de programación, para conocer el tipo de cada construcción del lenguaje.



Santana Gonzalez Jesus Salvador

## PARTE 5

### Análisis Sintáctico.

Tiene como función etiquetar cada uno de los componentes sintácticos que aparecen en la oración y analizar como las palabras se combinan para formar construcciones gramaticalmente correctas. El resultado de este proceso consiste en generar la estructura correspondiente a las categorías sintácticas formadas por cada una de las unidades léxicas que aparecen en la oración. Las gramáticas, tal como se muestran en la siguiente figura, están formadas por un conjunto de reglas cuales son:



43 O --> SN,  
SV SN --> Det,  
N SN --> Nombre Propio  
SV --> V,  
SN SV --> V SP --> Preposición,  
SN SN = sintagma nominal  
SV = sintagma verbal Det = determinante

El analizador léxico tiene como entrada el código fuente en forma de una sucesión de caracteres, el analizador sintáctico tiene como entrada los lexemas que le suministra el analizador léxico y su función es comprobar que están ordenados de forma correcta (dependiendo del lenguaje que queramos procesar) los dos analizadores suelen trabajar unidos e incluso el léxico suele ser una subrutina del sintáctico

### **Análisis Semántico.**

Es la extensión del análisis sintáctico para la comprensión de la redes semánticas y la base de conocimientos tanto para ver si tiene sentido y que este dirigida al lenguaje natural ya que es a eso a lo que se enfoca, para decirlo en otras palabras es darle significado asociado a las estructuras formales del lenguaje.

Este analizador computa información adicional necesaria para el procesamiento de un lenguaje una vez que se halla tomando bien un análisis completo y exacto del analizador sintáctico y genere resultados verídicos su objetivo primordial o principal del analizador semántico es que el programa analizado satisfaga las reglas requeridas por la especificación del lenguaje para garantizar su correcta ejecución.

### **Ventajas del análisis semántico**

- La descripción formal de la semántica de lenguajes de programación es la descripción rigurosa del significado o comportamiento de programas, el lenguaje de programación además de verificar propiedades de relación con pruebas de corrección o información relacionada con su ejecución.
- Ser utilizados como base para la implementación de procesadores de lenguaje.
- Facilitar la comprensión de los lenguajes por parte del programador y como mecanismo de comunicación entre el diseñador del lenguaje e implementador, la especificación semántica de un lenguaje como documento de referencia aclara el comportamiento del lenguaje y sus diversas construcciones.
- Revela posibles ambigüedades existentes.
- Crear implementaciones de procesadores de lenguajes o en documentos descriptivos del procesamiento del lenguaje natural.



El analizador semántico detecta la validez semántica de las sentencias aceptadas por el analizador sintáctico. El analizador semántico suele trabajar simultáneamente al analizador sintáctico y en estrecha cooperación se entiende por semántica como el conjunto de reglas que especifican el significado de cualquier sentencia sintácticamente correcta y escrita en un determinado lenguaje.

Las rutinas semánticas deben realizar la evaluación de los atributos de las gramáticas siguiendo las reglas semánticas asociadas a cada producción de la gramática. El análisis sintáctico es la fase en la que se trata de determinar el tipo de los resultados intermedios, comprobar que los argumentos que tiene un operador pertenecen al conjunto de los operadores posibles y si son compatibles entre sí, etc. En definitiva, comprobará que el significado de lo que se va leyendo es válido.

La salida “teórica” de la fase de análisis semántico sería un árbol semántico. Consiste en un árbol sintáctico en el que cada una de sus ramas ha adquirido el significado que debe tener. En el caso de los operadores polimorfismo (un único símbolo con varios significados), el análisis semántico determina cual es el aplicable.

Se compone de un conjunto de rutinas independientes, llamadas por los analizadores morfológicos y sintácticos. El análisis semántico utiliza como entrada el árbol sintáctico detectado por el análisis sintáctico para comprobar restricciones de tipo y otras limitaciones 52 semánticas y preparadas la generación de código. Las rutinas semánticas suelen hacer uso de una pila (la pila semántica) que contiene la información semántica asociada a los operandos en forma de registro semánticos.

## **Front end**

El Frontend hace uso de las tecnologías o lenguajes de estilo o programación del lado del cliente para la estructuración, el maquetado y la animación de los sitios web. Los lenguajes de estilo o programación a los que nos referimos son: HTML (lenguaje de marcas de hipertexto), CSS (hojas de estilo en cascada) y JavaScript, entre otros:

El HTML es un lenguaje de marcado que sirve para definir la estructura del contenido de tu web.

El CSS es un lenguaje de estilo que sirve para codificar la estructura creada por el HTML (darle color al texto, incluir márgenes, cambiar la tipografía del contenido...).



El JavaScript es un lenguaje de programación con el que puedes programar la interacción con el usuario.

Una vez aclarados estos tres términos, ¿qué es el Frontend? El Frontend es una tecnología que se encarga del diseño de una página web, es decir, se hace cargo de la estética, borradores, mockups (maqueta del diseño a escala o a tamaño real), interfaz, usabilidad de usuario, estructura, colores, tipografías y efectos. Asimismo, a través de JavaScript está la posibilidad de programar eventos, y validar formularios de contacto, entre otras funciones. En otras palabras, el Frontend es la parte del software encargada de interactuar con los visitantes de la web.

Es la parte de componente visible para el usuario. El front-end en diseño de software y desarrollo web hace referencia a la visualización del usuario navegante o, dicho de otra manera, es la parte que interactúa con los usuarios.

El Frontend abarca aquellas tecnologías que hacen referencia exclusivamente de la comodidad visual del usuario. Estiliza la página web mediante las técnicas que ofrece la User Experience (experiencia del usuario). También conocerá acerca de diseño web para que la estructura del sitio facilite una visualización ordenada y una comodidad para quien esté navegando por la página. Además, se ocupará de generar una web intuitiva para el usuario.

Estas tecnologías se generalizan en los siguientes lenguajes de programación:

- HTML: Encargado de ordenar el contenido de un sitio web.
- CSS: Forma parte del diseño gráfico y se ocupa de la creación y estructuras de los documentos webs.
- JAVA SCRIPT.:Permite la creación de actividades complejas en el desarrollo web como actualizaciones instantáneas, mapas, infografías, 3D, etcétera.

El programador encargado del frontend manejará solo estas 3 tecnologías, conociendo también a la perfección acerca del backend aunque no trabaje con él, ya que están íntimamente relacionados y uno necesita del otro.



En pocas palabras, esta tecnología web se encargará de optimizar visualmente la página, estructurarla cómodamente a los ojos del usuario y mantener una adecuada estética del sitio para que la interacción en él (consulta, solicitud, clics) sea correcta y eficaz. Por ello, una de las cualidades más valiosas para obtener una web responsive es la creatividad para poder diseñar sitios llamativos y atractivos que sean óptimos para todo tipo de dispositivos y resoluciones.

Los componentes que se encuentran en la parte frontal del sistema son los siguientes:

- pruebas de usabilidad y accesibilidad;
- lenguajes de diseño y marcado como HTML, CSS y JavaScript;
- diseño gráfico y herramientas de edición de imágenes;
- posicionamiento en buscadores o SEO;
- rendimiento web y compatibilidad del navegador.

- A configuration of an LR parser is:  
 $(s_0 X_1 s_2 X_2 \dots X_m s_m, a_1 a_{i+1} \dots a_n \$)$ , where,  
**stack**                      **unexpended input**  
 $s_0, s_1, \dots, s_m$ , are the states of the parser, and  $X_1, X_2, \dots, X_m$ ,  
are grammar symbols (terminals or nonterminals)
- Starting configuration of the parser:  $(s_0, a_1 a_2 \dots a_n \$)$ ,  
where,  $s_0$  is the initial state of the parser, and  $a_1 a_2 \dots a_n$  is  
the string to be parsed
- Two parts in the parsing table: *ACTION* and *GOTO*
  - The *ACTION* table can have four types of entries: **shift**,  
**reduce**, **accept**, or **error**
  - The *GOTO* table provides the next state information to be  
used after a *reduce* move





## Parte 6

### Analizador Léxico, Sintáctico y Semántico

Los ordenadores son una mezcla equilibrada de Software y Hardware

Los compiladores son programas de computadora que traducen de un lenguaje a otro un lenguaje escrito en lenguaje fuente y produce un programa equivalente escrito en lenguaje objeto

Un compilador se compone internamente de varias etapas o faces que realizan operaciones lógicas y estas son:

#### Analizador léxico

Lee la secuencia de caracteres de izquierda a derecha del programa fuente y agrupa la secuencia de caracteres en unidades con significado propio (componentes léxicos o tokens)

Las palabras clave identificadores, operadores, constantes numéricas, signos de puntuación como separadores de sentencia, llaves, parentesis, etcétera. son diversas clasificaciones de componentes léxicos.

#### Análisis léxico (Scanner)

Scanner tiene las funciones de leer el programa fuente como un archivo de caracteres y dividirlo en tokens. Los tokens son las palabras reservadas de un lenguaje, secuencia de caracteres que representa una unidad de información en el programa fuente. En cada caso un token representa un cierto patrón de caracteres que el analizador léxico reconoce, o ajusta desde el inicio de los caracteres de entrada. De tal manera es necesario generar un mecanismo computacional que nos permita identificar el patrón de transición entre los caracteres de entrada, generando tokens, que posteriormente serán clasificados. Este mecanismo es posible crearlo a partir de un tipo específico de maquina de estados llamado autómatas finitos.

#### Representación de un Analizador léxico

Los componentes léxicos se representan:

1. Palabras reservadas: if, while, do, ...
2. Identificadores: variables, funciones, tipos definidos por el usuario, etiquetas, ...
3. Operadores: =, >, <, >=, <=, +, \*, ...
4. Símbolos especiales: ;, ( ), { }, ...
5. Constantes numéricas. literales que representan valores enteros y flotantes.
6. Constantes de carácter: literales que representan cadenas de caracteres.

¿Qué es un analizador léxico?





Se encarga de buscar los componentes léxicos o palabras que componen el programa fuente, según unas reglas o patrones. La entrada del analizador léxico podemos definirla como una secuencia de caracteres.

El analizador léxico tiene que dividir la secuencia de caracteres en palabras con significado propio y después convertirlo a una secuencia de terminales desde el punto de vista del analizador sintáctico, que es la entrada del analizador sintáctico. El analizador léxico reconoce las palabras en función de una gramática regular de manera que sus NO TERMINALES se convierten en los elementos de entrada de fases posteriores.

### Análisis sintáctico

determina si la secuencia de componentes léxicos sigue la sintaxis del lenguaje y obtiene la estructura jerárquica del programa en forma de árbol, donde los nodos son las construcciones de alto nivel del lenguaje.

Se determina las relaciones estructurales entre los componentes léxicos esto es semejante a realizar el análisis gramatical sobre una frase en el lenguaje natural. La estructura sintáctica se define mediante las gramáticas independientes del contexto

### ¿Qué es el analizador sintáctico?

Es la fase del analizador que se encarga de chequear el texto de entrada en base a una gramática dada. Y en caso de que el programa de entrada sea válido, suministra el árbol sintáctico que lo reconoce. En teoría, se supone que la salida del analizador sintáctico es alguna representación del árbol sintáctico que reconoce la secuencia de tokens suministrada por el analizador léxico. En la práctica, el analizador sintáctico también hace:

- Acceder a la tabla de símbolos (para hacer parte del trabajo del analizador semántico).
- Chequeo de tipos (del analizador semántico).
- Generar código intermedio.

### Análisis Semántico

El análisis semántico dota de un significado coherente a lo que hemos hecho en el análisis sintáctico.

El chequeo semántico se encarga de que los tipos que intervienen en las expresiones sean compatibles o que los parámetros reales de una función sean coherentes con los parámetros formales

### Funciones principales

Identificar cada tipo de instrucción y sus componentes

Completar la Tabla de Símbolos

Realizar distintas comprobaciones y validaciones:

Comprobaciones de tipos.

Comprobaciones del flujo de control.



Comprobaciones de unicidad.

Ejemplo de un Analizador-Lexico-Sintactico-Semantico:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- You may freely edit this file. See commented blocks below for -->
<!-- some examples of how to customize the build. -->
<!-- (If you delete it and reopen the project it will be recreated.) -->
<!-- By default, only the Clean and Build commands use this build script. -->
<!-- Commands such as Run, Debug, and Test only use this build script if -->
<!-- the Compile on Save feature is turned off for the project. -->
<!-- You can turn off the Compile on Save (or Deploy on Save) setting -->
<!-- in the project's Project Properties dialog box.-->
<project name="CompiladorFinal" default="default" basedir=".">
  <description>Builds, tests, and runs the project CompiladorFinal.</description>
  <import file="nbproject/build-impl.xml"/>
  <!--
```

There exist several targets which are by default empty and which can be used for execution of your tasks. These targets are usually executed before and after some main targets. They are:

- pre-init: called before initialization of project properties
- post-init: called after initialization of project properties
- pre-compile: called before javac compilation
- post-compile: called after javac compilation
- pre-compile-single: called before javac compilation of single file
- post-compile-single: called after javac compilation of single file
- pre-compile-test: called before javac compilation of JUnit tests
- post-compile-test: called after javac compilation of JUnit tests
- pre-compile-test-single: called before javac compilation of single JUnit test
- post-compile-test-single: called after javac compilation of single JUnit test
- pre-jar: called before JAR building
- post-jar: called after JAR building
- post-clean: called after cleaning build products

(Targets beginning with '-' are not intended to be called on their own.)

Example of inserting an obfuscator after compilation could look like this:

```
<target name="-post-compile">
  <obfuscate>
    <fileset dir="${build.classes.dir}"/>
  </obfuscate>
</target>
```

For list of available properties check the imported nbproject/build-impl.xml file.

Another way to customize the build is by overriding existing main targets.





The targets of interest are:

- init-macrodef-javac: defines macro for javac compilation
- init-macrodef-junit: defines macro for junit execution
- init-macrodef-debug: defines macro for class debugging
- init-macrodef-java: defines macro for class execution
- do-jar: JAR building
- run: execution of project
- javadoc-build: Javadoc generation
- test-report: JUnit report generation

An example of overriding the target for project execution could look like this:

```
<target name="run" depends="CompiladorFinal-impl.jar">
  <exec dir="bin" executable="launcher.exe">
    <arg file="${dist.jar}"/>
  </exec>
</target>
```

Notice that the overridden target depends on the jar target and not only on the compile target as the regular run target does. Again, for a list of available properties which you can use, check the target you are overriding in the nbproject/build-impl.xml file.

-->

</project>

Ejemplo frontend:

nombre-proyecto/

```
|__ src
|  |__ scss
|  |  |__ style.scss
|  |  |__ inc
|  |    |__ mixins.scss
|  |    |__ normalize.scss
|  |    |__ colors.scss
|  |    |__ variables.scss
|  |    |__ components.scss
|  |
|  |__ jade
|  |  |__ page.jade
|  |  |__ inc
|  |    |__ mixins.jade
|  |    |__ variables.jade
|  |  |__ template
|  |    |__ templatenamename.jade
|  |
```



```
|   |__ js
|   |   |__ functions.js
|   |
|   |__ images
|       |__ sprites
|
|__ gruntfile.js
|__ package.json
|__ bower.json
|__ .editorconfig
|__ .gitignore
|__ .htmlhintc
|__ .jshintc
|
|__ dist
|   |__ page.html
|   |__ assets
|       |__ css
|           |__ style.css
|           |__ libs
|               |__ anyexternallib.css
|       |__ js
|           |__ functions.js
|           |__ libs
|               |__ jquery-1.11.3.min.js
|               |__ modernizr.js
|               |__ detectizr.js
|               |__ lt-ie-9.min.js
|               |__ anyexternallib.js
|       |__ images
|           |__ sprites.png
|
|__ build
|   |__ page.html
|   |__ assets
|       |__ css
|           |__ styles.min.css
|       |__ js
|           |__ functions.min.js
|           |__ libs
|               |__ jquery-1.11.3.min.js
|               |__ modernizr-detectizr.min.js
```



```
|
|
|
|__ ie.min.js
|__ plugins.min.js
|
|__ images
|__ sprites.png
```

Santana Gonzalez Jesus Salvador

### 3.2.2 "Gramáticas independientes del contexto (Context-Free Grammars)"

Una gramática  $G = \langle \Sigma, \Gamma, S, \rightarrow \rangle$  es independiente del contexto si todas las reglas de producción tienen una de las dos siguientes formas

$A \rightarrow \alpha \quad A \rightarrow \beta$ ,

donde  $A \in \Gamma$ .

CFG designará el conjunto de gramáticas independientes del contexto.

El lenguaje generado por  $G$  se denotará con  $L(G)$ .

Un lenguaje  $L$  es independiente del contexto si existe una gramática  $G \in \text{CFG}$  tal que

$L = L(G)$ .

CFL es el conjunto de lenguajes independientes del contexto

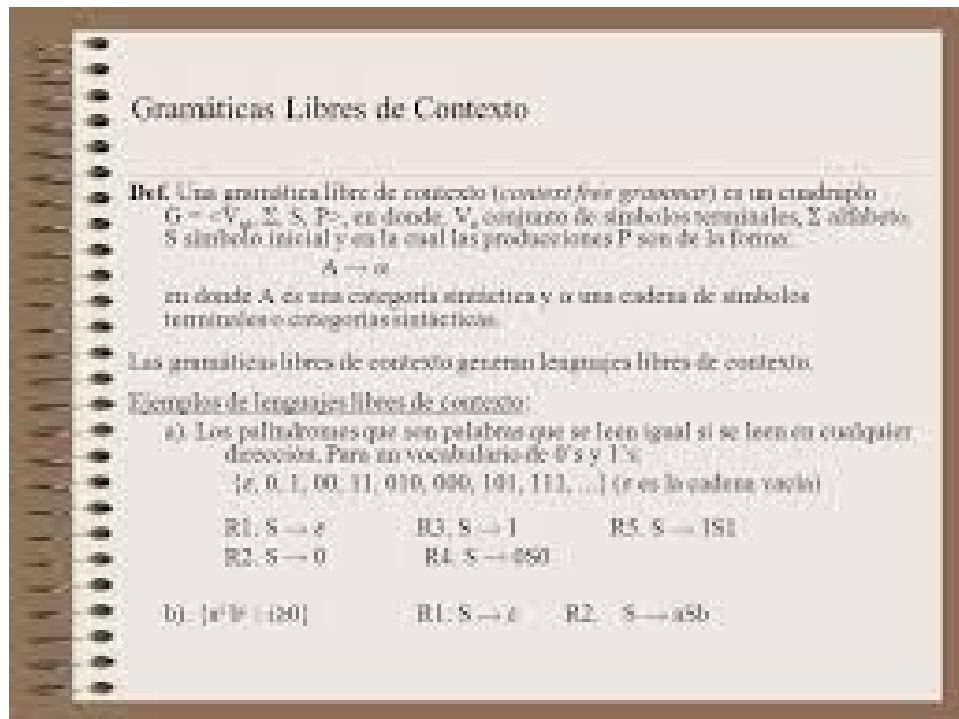
Lenguaje de palindromos. Sea  $GP = \langle \{a, b\}, \{S\}, S, \rightarrow P_i \rangle$ , donde  
 $S \rightarrow P \mid aSa \mid bSb$ .

2 Lenguaje de paréntesis. Sea  $GD = \langle \{(\cdot), \cdot\}, \{S\}, S, \rightarrow D_i \rangle$ , con las siguientes reglas

$S \rightarrow D(S) \mid SS \mid \cdot$ .

En la teoría del lenguaje formal, una gramática libre de contexto (CFG) es una gramática formal en la que cada regla de producción tiene la forma  $A \rightarrow \alpha$ , donde  $A$  es un solo símbolo no terminal y  $\alpha$  es una cadena de terminales y / o no terminales (puede estar vacío). Una gramática formal se considera "libre de contexto" cuando sus reglas de producción se pueden aplicar independientemente del contexto de un no terminal. Independientemente de los símbolos que lo rodeen, el no terminal único del lado izquierdo siempre se puede reemplazar por el lado derecho. Esto es lo que lo distingue de una gramática sensible al contexto. Una gramática formal es esencialmente un conjunto de reglas de producción que describen todas las cadenas posibles en un lenguaje formal dado. Las reglas de producción son reemplazos simples. Por ejemplo, la primera regla de la imagen, reemplaza  $A$  con  $\alpha$ . Puede haber varias reglas de reemplazo para un símbolo no terminal dado. El lenguaje generado por una gramática es el conjunto de todas las cadenas de

símbolos terminales que pueden derivarse, mediante aplicaciones repetidas de reglas, de algún símbolo no terminal particular ("símbolo de inicio"). Los símbolos no terminales se utilizan durante el proceso de derivación, pero es posible que no aparezcan en su cadena de resultado final.



Una gramática regular es una 4-tupla  $G=(\Sigma,N,S,P)$ , donde  $\Sigma$  es el alfabeto,  $N$  es una colección de símbolos no terminales,  $S$  es un símbolo no terminal llamado símbolo inicial y  $P$  es un conjunto de reglas de sustitución, llamadas producciones de la forma  $A \rightarrow w$  donde  $A \in N$  y  $w \in (\Sigma \cup N)^*$  satisfaciendo:

- 1)  $w$  contiene un no terminal como máximo.
- 2) Si  $w$  contiene un no terminal, entonces es el símbolo que está en el extremo derecho de  $w$ .

El lenguaje generado por la gramática se representa como  $L(G)$ .

Sea  $L$  el lenguaje regular reconocido por un AFD  $M = (\Sigma, Q, s, F, \delta)$ . La gramática regular que genera el lenguaje  $L$  será  $G = (\Sigma, N, S, P)$  tal que:  $\Sigma = \Sigma$ ,  $N = Q$ ,  $S = s$ ,  $P = \{ q \rightarrow ap \mid \delta(q, a) = p \} \cup \{ q \rightarrow \epsilon \mid q \in F \}$ . Todas aquellas palabras que reconoce el AFD  $M$ , pueden ser generadas a partir de las producciones de  $G$ .

Una Gramática Independiente del Contexto (GIC) es una 4-tupla  $G = (\Sigma, N, S, P)$



donde  $\Sigma$  es el alfabeto (conjunto de terminales),  $N$  es la colección finita de los no terminales,  $S$  es un no terminal llamado símbolo inicial y  $P \subseteq N \times (N \cup \Sigma)^*$  es un conjunto de producciones.

El lenguaje generado por la GIC se denota  $L(G)$  y se llama Lenguaje Independiente del Contexto (LIC).

Relación GR y GIC:

$GR \subseteq GIC$

$LR \subseteq LIC$

Sea  $L$  el lenguaje generado por la gramática regular  $G = (\Sigma, N, S, P)$ . El AFN que reconoce el lenguaje  $L$  será  $M = (\Sigma, Q, s, F, \Delta)$  tal que:

$Q = N \cup \{f\}$ , siendo  $f$  un símbolo nuevo

$s = S$

$F = \{f\}$

$\Delta$  se definen:

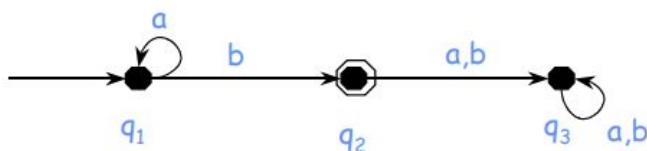
- Si  $A \rightarrow \sigma_1 \dots \sigma_n B$  con  $A$  y  $B$  no terminales, entonces se añadirán a  $Q$  los nuevos estados  $q_1, q_2, \dots, q_{n-1}$  y las transformaciones siguientes:

$\Delta(A, \sigma_1 \dots \sigma_n) = \Delta(q_1, \sigma_2 \dots \sigma_n) = \dots = \Delta(q_{n-1}, \sigma_n) = B$

- Si  $A \rightarrow \sigma_1 \dots \sigma_n$  con  $A$  no terminal, entonces se añadirán a  $Q$  los nuevos estados  $q_1, q_2, \dots, q_{n-1}$  y las transformaciones siguientes:

$\Delta(A, \sigma_1 \dots \sigma_n) = \Delta(q_1, \sigma_2 \dots \sigma_n) = \dots = \Delta(q_{n-1}, \sigma_n) = f$

- Ejemplo: Sea el AFD  $M$ ,



Santana Gonzalez Jesus Salvador

Gramáticas independientes del contexto (context-free grammars):

usan reglas que predicen las palabras que pueden posiblemente seguir a la última palabra reconocida, reduciendo el número de palabras candidatas a evaluar para reconocer la siguiente pronunciación del speaker (hablante o fichero de voz).



Las reglas están formadas por dos tipos de símbolos: palabras y directrices.  
Ejemplo: =ALT(SEQ("Jesús", "Moreno"), "Miguel Alonso") que significa algo como "lo que en la categoría que se puede dar como válido al reconocer el habla es una de las siguientes alternativas: la secuencia Jesús y detrás Moreno o Miguel Alonso"  
Estas gramáticas usan "pedazos" (chunks) de información para comunicarse con el motor de reconocimiento: Words, Rules, Exported Rules, Imported Rules, Run-Time Lists.

Una Gramática independientes del contexto (GIC) es una gramática formal en la que cada regla de producción es de la forma:

$\text{Exp} \rightarrow x$

Donde Exp es un símbolo no terminal y x es una cadena de terminales y / o no terminales. El término independiente del contexto se refiere al hecho de que el no terminal Exp puede siempre ser sustituido por x sin tener en cuenta el contexto en el que ocurre. Un lenguaje formal es independiente de contexto si hay una gramática libre de contexto que lo general, este tipo de gramática fue creada por Backus-Naur y se utiliza para describir la mayoría de los lenguajes de programación.

Una GIC está compuesta por 4 elementos:

1. Símbolos terminales (elementos que no generan nada)
2. No terminales (elementos del lado izquierdo de una producción, antes de la flecha "->")
3. Producciones (sentencias que se escriben en la gramática)
4. Símbolo inicial (primer elemento de la gramática)

Ejemplo 1: Teniendo un lenguaje que genera expresiones de tipo:

$9 + 5 - 2$

Para determinar si una GIC esta bien escrita se utilizan los arboles de analisis sintáctico , así:

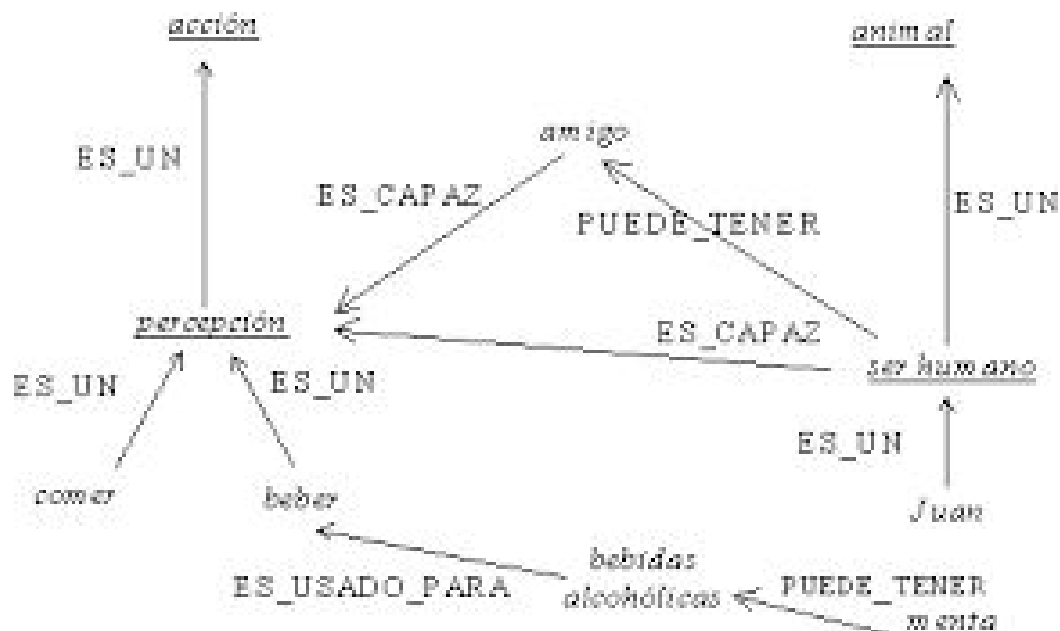
Producciones:

lista -> lista + digito

lista -> lista - digito

lista -> digito

digito -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



Las Gramáticas Libres de Contexto ( ' Context-Free Languages) o CFLs jugaron un papel central en lenguaje natural desde los 50's y en los compiladores desde los 60's

Las Gramáticas Libres de Contexto forman la base de ' la sintaxis BNF

Son actualmente importantes para XML y sus DTDs (document type definition)

Una gramatica libre de contexto se define con '  $G = (V, T, P, S)$  donde:

- V es un conjunto de variables
- T es un conjunto de terminales
- P es un conjunto finito de producciones de la forma  $A \rightarrow \alpha$ , donde A es una variables y  $\alpha \in (V \cup T)^*$
- S es una variable designada llamada el s'imbolo inicial

Ejemplo:

$G_{pal} = (\{P\}, \{0, 1\}, A, P)$ , donde  $A = \{P \rightarrow , P \rightarrow 0, P \rightarrow 1, P \rightarrow 0P0, P \rightarrow 1P1\}$ .

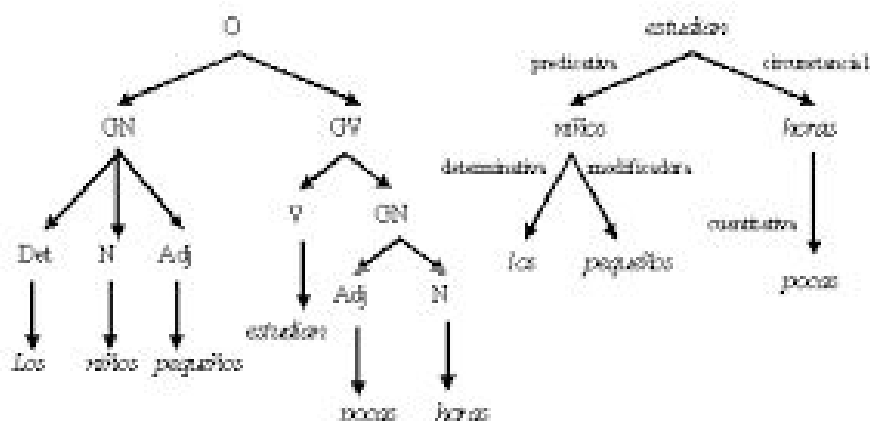
• Muchas veces se agrupan las producciones con la misma cabeza, e.g.,  $A = \{P \rightarrow |0|1|0P0|1P1\}$ .

• Ejemplo: Expresiones regulares sobre  $\{0, 1\}$  se pueden definir por la gramatica: '  $G_{regex} = (\{E\}, \{0, 1\}, A, E)$ , donde  $A = \{E \rightarrow 0, E \rightarrow 1, E \rightarrow E.E, E \rightarrow E + E, E \rightarrow E^*, E \rightarrow (E)\}$ .

Estas gramáticas, conocidas también como gramáticas de tipo 2 o gramáticas independientes del contexto, son las que generan los lenguajes libres o independientes del contexto.



Los lenguajes libres del contexto son aquellos que pueden ser reconocidos por un autómata de pila determinista o no determinístico.



A) Árbol de constituyentes

B) Árbol de dependencias

Las gramáticas libres del contexto se escriben, frecuentemente, utilizando una notación conocida como BNF (Backus-Naur Form). BNF es la técnica más común para definir la sintaxis de los lenguajes de programación.

En esta notación se deben seguir las siguientes convenciones:

- los no terminales se escriben entre paréntesis angulares < >
- los terminales se representan con cadenas de caracteres sin paréntesis angulares
- el lado izquierdo de cada regla debe tener únicamente un no terminal (ya que es una gramática libre del contexto) - el símbolo ::=, que se lee "se define como" o "se reescribe como", se utiliza en lugar de →
- varias producciones del tipo ::= ::= . . . ::= se pueden escribir como ::= ...

### Árbol de derivación

Un árbol de derivación permite mostrar gráficamente cómo se puede derivar cualquier cadena de un lenguaje a partir del símbolo distinguido de una gramática que genera ese lenguaje.

Un árbol es un conjunto de puntos, llamados nodos, unidos por líneas, llamadas arcos.

Un arco conecta dos nodos distintos.

Para ser un árbol un conjunto de nodos y arcos debe satisfacer ciertas propiedades:

- hay un único nodo distinguido, llamado raíz (se dibuja en la parte superior) que no tiene arcos incidentes.
- todo nodo *c* excepto el nodo raíz está conectado con un arco a otro nodo *k*, llamado el padre de *c* (*c* es el hijo de *k*). El padre de un nodo, se dibuja por encima del nodo.
- todos los nodos están conectados al nodo raíz mediante un único camino.





- los nodos que no tienen hijos se denominan hojas, el resto de los nodos se denominan nodos interiores. El árbol de derivación tiene las siguientes propiedades:
- el nodo raíz está rotulado con el símbolo distinguido de la gramática;
- cada hoja corresponde a un símbolo terminal o un símbolo no terminal;
- cada nodo interior corresponde a un símbolo no terminal.

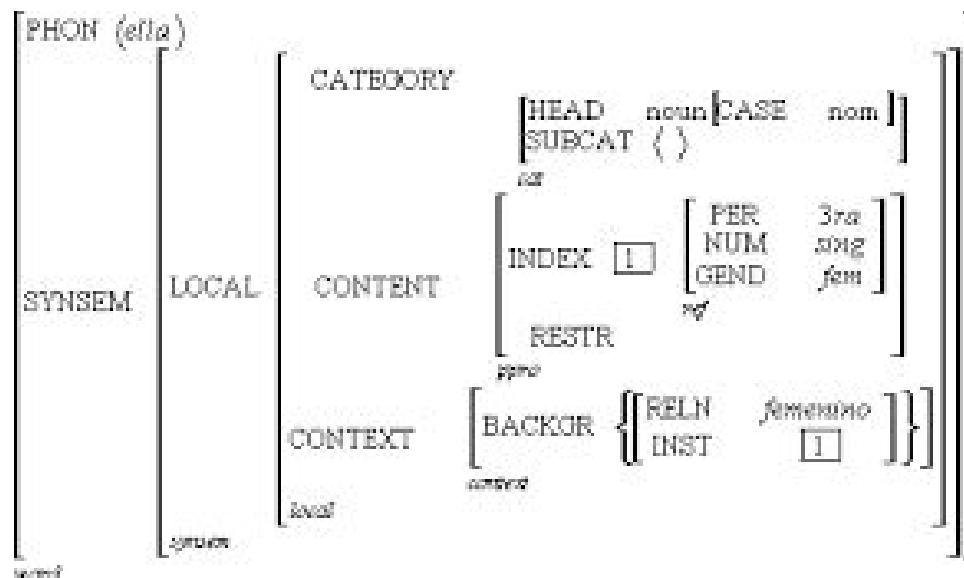
### Gramáticas ambiguas

Una gramática es ambigua si permite construir dos o más árboles de derivación distintos para la misma cadena. Por lo tanto, para demostrar que una gramática es ambigua lo único que se necesita es encontrar una cadena que tenga más de un árbol de derivación.

Una gramática en la cual, para toda cadena generada  $w$ , todas las derivaciones de  $w$  tienen el mismo árbol de derivación es no ambigua. En algunos casos, dada una gramática ambigua, se puede encontrar otra gramática que produzca el mismo lenguaje pero que no sea ambigua.

Si todas las gramáticas independientes del contexto para un lenguaje son ambiguas, se dice que el lenguaje es un lenguaje independiente del contexto inherentemente ambiguo.

Por ejemplo, el lenguaje  $L = \{ a^i b^j c^k / i = j \text{ ó } j = k \}$



Santana Gonzalez Jesus Salvador

### Semana 14

Una representación intermedia es una estructura de datos que representa al programa fuente durante el proceso de la traducción a código objeto. Hasta ahora hemos usado el árbol de análisis sintáctico como representación intermedia, junto con la tabla de símbolos que contenía información sobre los nombres (variables, constantes, tipos y funciones) que



aparecían en el programa fuente. Aunque el árbol de análisis sintáctico es una representación válida, no se parece ni remotamente al código objeto, en el que solo se emplean saltos a direcciones en memoria en vez de construcciones de alto nivel, como sentencias if-then-else. Es necesario generar una nueva forma de representación intermedia. A esta representación intermedia, que se parece al código objeto pero que sigue siendo independiente de la máquina, se le llama código intermedio. El código intermedio puede tomar muchas formas. Todas ellas se consideran como una forma de linearización del árbol sintáctico, es decir, una representación del árbol sintáctico de forma secuencial. El código intermedio más habitual es el código de 3- direcciones. El código de tres direcciones es una secuencia de proposiciones de la forma general  $x = y \text{ op } z$  donde  $p$  representa cualquier operador;  $x, y, z$  representan variables definidas por el programador o variables temporales generadas por el compilador.  $y, z$  también pueden representar constantes o literales.  $op$  representa cualquier operador: un operador aritmético de punto fijo o flotante, o un operador lógico sobre datos booleanos. No se permite ninguna expresión aritmética compuesta, pues sólo hay un operador en el lado derecho. Por ejemplo,  $x+y*z$  se debe traducir a una secuencia, donde  $t1, t2$  son variables temporales generadas por el compilador.  $t1 = y * z$   $t2 = x + t1$  Las expresiones compuestas y las proposiciones de flujo de control se han de descomponer en proposiciones de este tipo, definiendo un conjunto suficientemente amplio de operadores. Se le llama código de 3-direcciones porque cada proposición contiene, en el caso general, tres direcciones, dos para los operandos y una para el resultado. (Aunque aparece el nombre de la variable, realmente corresponde al puntero a la entrada de la tabla de símbolos de dicho nombre). El código de tres direcciones es una representación linealizada (de izquierda a derecha) del árbol sintáctico en la que los nombres temporales corresponden a los nodos internos. Como estos nombres temporales se representan en la memoria no se especifica más información sobre ellos en este tipo de código. Normalmente se asignan directamente a registros o se almacenan en la tabla de símbolos.  $2*a+b-3$ .

La forma de código de 3-direcciones es insuficiente para representar todas las construcciones de un lenguaje de programación (saltos condicionales, saltos incondicionales, llamadas a funciones, bucles, etc), por tanto es necesario introducir nuevos operadores. El conjunto de proposiciones (operadores) debe ser lo suficientemente rico como para poder implantar las operaciones del lenguaje fuente. Las proposiciones de 3-direcciones van a ser en cierta manera análogas al código ensamblador. Las proposiciones pueden tener etiquetas simbólicas y existen instrucciones para el flujo de control (goto). Una etiqueta simbólica representa el índice de una proposición de 3-direcciones en la lista de instrucciones.

Las proposiciones de 3-direcciones más comunes:

1. Proposiciones de la forma  $x = y \text{ op } z$  donde  $op$  es un operador binario aritmético, lógico o relacional.
2. Instrucciones de la forma  $x = op y$ , donde  $op$  es un operador unario (operador negación lógico, operadores de desplazamiento o conversión de tipos).
3. Proposiciones de copia de la forma  $x = y$ , donde el valor de  $y$  se asigna a  $x$ .
4. Salto incondicional goto etiq. La instrucción con etiqueta etiq es la siguiente que se ejecutará.
5. Saltos condicionales como if false x goto etiq.



6. param x y call f para apilar los parámetros y llamadas a funciones (los procedimientos se consideran funciones que no devuelven valores). También return y, que es opcional, para devolver valores. Código generado como parte de una llamada al procedimiento p(x 1,x 2,...,x n). param x1 param x2 ... param xn call p,n

7. Asignaciones con índices de la forma  $x = y[i]$ , donde se asigna a x el valor de la posición en i unidades de memoria más allá de la posición y. O también  $x[i] = y$ .

8. Asignaciones de direcciones a punteros de la forma  $x = \&y$  (el valor de x es la dirección de y),  $x = *y$  (el valor de x se iguala al contenido de la dirección indicada por el puntero y) o  $*x = y$  (el objeto apuntado por x se iguala al valor de y).

Ejemplo. Consideremos el código que calcula el factorial de un número. La tabla 7.1 muestra el código fuente y el código de 3- direcciones. Existe un salto condicional if false que se usa para traducir las sentencias de control if-then, repeat-until que contiene dos direcciones: el valor condicional de la expresión y la dirección de salto.

La proposición label sólo tiene una dirección.

Las operaciones de lectura y escritura, read, write, con una sola dirección.

Y una instrucción de parada halt que no tiene direcciones.

read x; read x

if 0<x then t1 = 0 < x

fact:=1; if false t1 goto L1

repeat fact=1

fact:=fact\*x; label L2

x:=x-1; t2=fact \* x

until x=0; fact=t2

write fact; t3=x-1

end; x=t3

t4=x==0

if false t4 goto L2

write fact

label L1

halt

Santana Gonzalez Jesus Salvador