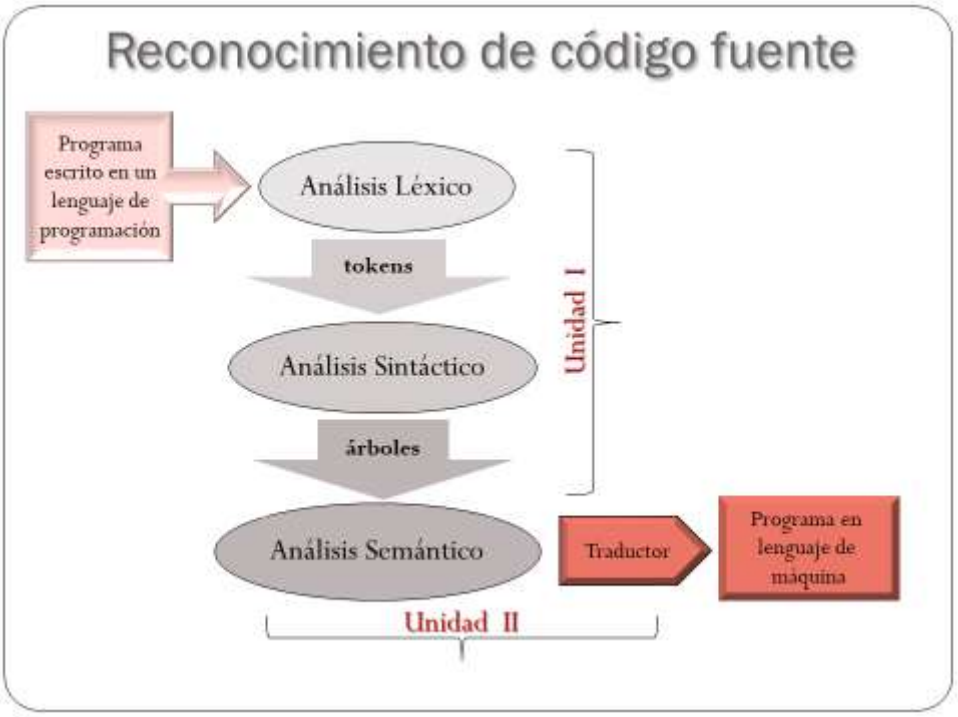
FILMINAS COMENTADAS
SINTÁXIS Y SEMÁNTICA DE LENGUAJES

INTRODUCCIÓN

El presente documento, toma como hilo conductor las filminas de clase e incluye notas y ejemplos que pretenden explicar sintéticamente los principales contenidos de la asignatura. Es una guía, que debe ser ampliada luego con la lectura bibliográfica, el material digital complementario propuesto, la resolución de trabajos prácticos y la indagación personal y autónoma por parte de los alumnos.

ETAPAS DE ANALISIS

Recordemos ahora la filmina en la cual mencionamos brevemente las tres primeras fases del proceso de traducción (Análisis Léxico, Sintáctico y Semántico). En los apartados anteriores hemos estado revisando conceptos, tales como alfabeto, lenguajes, gramáticas, expresiones regulares y autómatas, que explicamos tenían estrecha relación con la implementación de estos procesos. Es decir con el cómo se resuelven estas etapas mediante un programa de computación: el traductor.



Antes de empezar, propongamos la siguiente analogía. Supongamos que deseamos traducir la siguiente frase del castellano al inglés.

"Ese hombre cuenta"

Ese → adjetivo demostrativo

hombre → sustantivo

cuenta -> verbo

Lo primero que debemos hacer es verificar que la frase está formada por palabras válidas en el lenguaje castellano. Es decir que el léxico usado es válido (***Análisis Léxico***). Si la frase fuera

"Ese ombre cuenta"

Ese -> adjetivo demostrativo

ombre → ¿?

Entonces tenemos un problema léxico. La palabra *ombre*, no pertenece al léxico del lenguaje castellano.

Lo siguiente es verificar que la frase está bien formada de acuerdo a la sintáxis del lenguaje.

"Ese hombre cuenta"

sujeto predicado

Esta será la función del ***Análisis Sintáctico***. Si escribiéramos la frase como

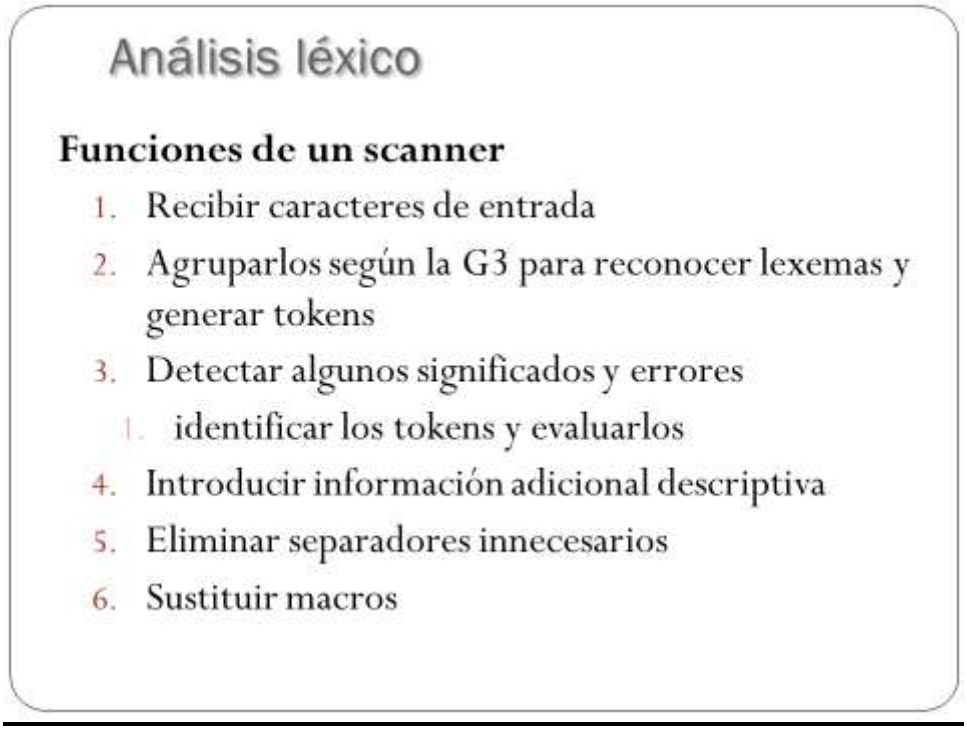
"cuenta hombre Ese"

El léxico es correcto, pero tenemos un problema sintáctico. La frase no está bien formada de acuerdo a la gramática del lenguaje.

Por último, antes de poder traducirla al inglés, debemos saber con exactitud qué significa la frase en castellano. Esta es la etapa de ***Análisis Semántico***. En nuestro ejemplo tenemos un problema semántico, porque existe ambigüedad semántica respecto del verbo cuenta. Puede significar que el hombre está realizando la acción de contar o que el hombre resulta importante o valioso para otro.

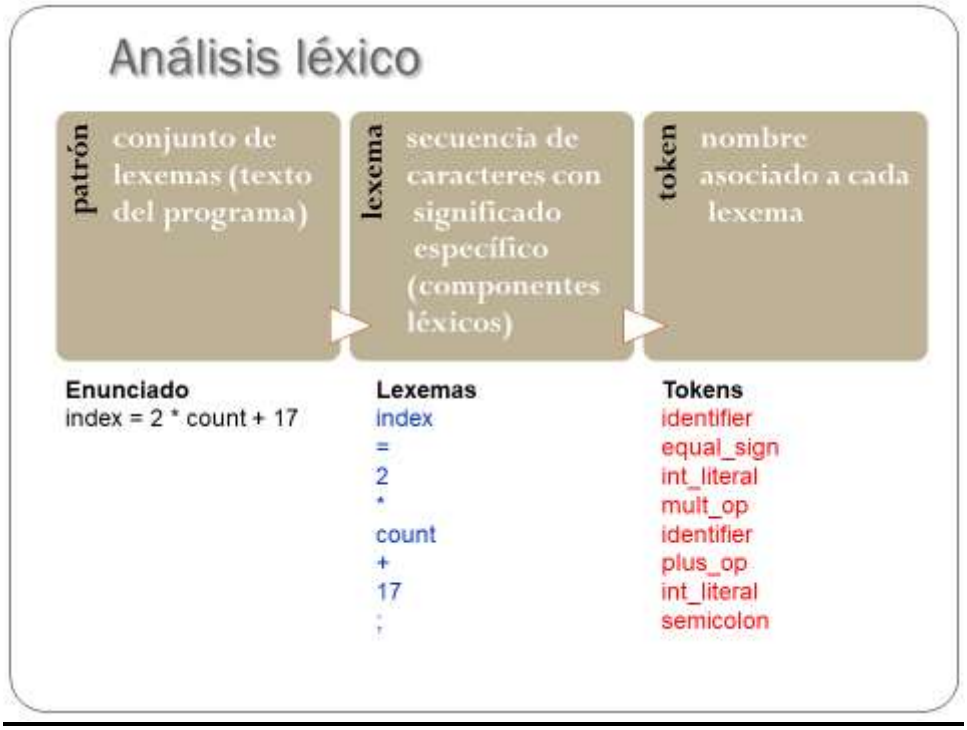
Con este ejemplo en mente, y contando con los conocimientos previamente adquiridos, trataremos ahora de explicar en qué consiste y cómo se implementa cada una de estas etapas en un programa traductor.

ANALISIS LÉXICO



Inicialmente el código fuente de un programa que es entregado al traductor, es un flujo de bytes sin estructura ni sentido alguno. Este flujo de bytes es entregado como input al analizador léxico (o scanner). El scanner deberá entonces estructurar y reconocer en los caracteres de entrada un flujo de código tokenizado (léxicamente válido o en su defecto encontrar errores léxicos).

Expliquemos este proceso siguiendo el ejemplo que nos presenta la siguiente filmina.



Supongamos que se entrega la sentencia *index = 2 * count + 17* como entrada al traductor. El primer trabajo que realiza el scanner, es reconocer que palabras o cadenas están presentes en la sentencia. Así individualiza las palabras *"index", "=", "2", "*", "count", "+", "17" y ";"* a las que llamaremos **Lexemas.**

Por otra parte, un **token**, es una categoría léxica en un lenguaje (un tipo de palabra válida en un lenguaje). En el lenguaje castellano, por ejemplo, la palabra *"casa"* es un lexema que puede ser ubicada en la categoría léxica de *sustantivo* (el token), la palabra *"correr"* correspondería al token *verbo*. De igual forma, serían tokens un artículo, un punto, etc. Si volvemos sobre nuestra analogía tendríamos:

Lexema	***Token***
Ese	*Adjetivo demostrativo*
Hombre	*Sustantivo*
Cuenta	*Verbo*

En la filmina son tokens o categorías léxicas en el lenguaje de alto nivel: un identificador (*identifier*), un signo de asignación (*equal sign*), una constante literal entera (*int_literal*), etc.

Teniendo en cuenta que el léxico de los lenguajes de programación está definido mediante gramáticas del tipo 3 en la jerarquía de Chomsky o G3 (o lo que es equivalente, a partir de expresiones regulares), y que para cada G3 es posible definir un $AFD_{mínimo}$; entonces, podemos contar con un autómata para cada categoría léxica o token del lenguaje, capaz de reconocer si una palabra o cadena de entrada (lexema) pertenece o no a esa categoría (token).

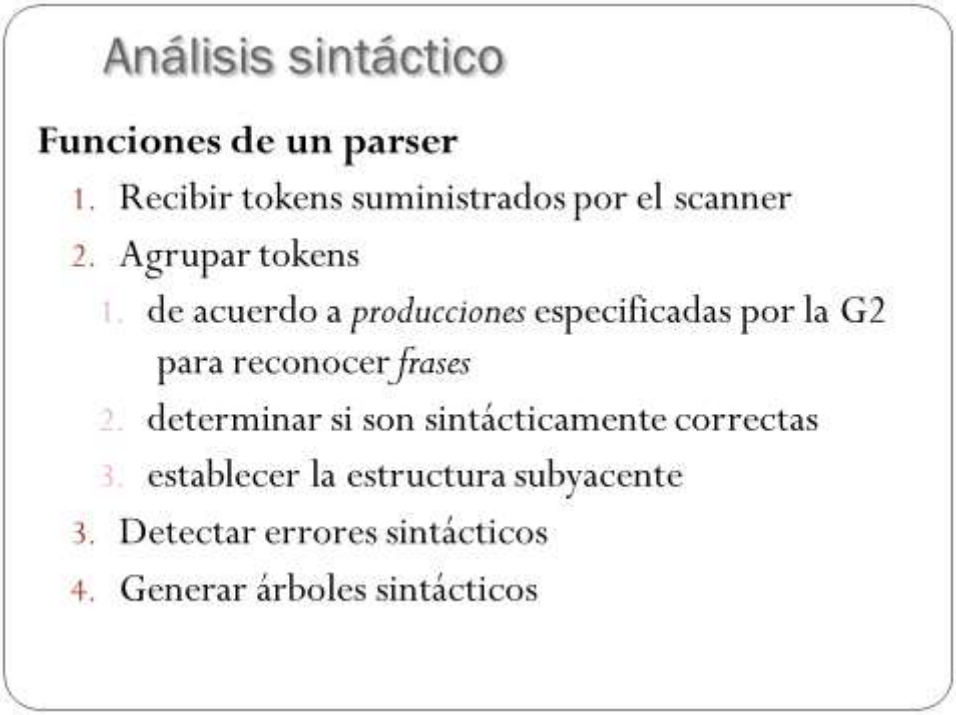
Así, el scanner estará compuesto de un conjunto de AFD encargados de tomar cada lexema reconocido y evaluar si es válido o no de acuerdo a alguna categoría léxica en

el lenguaje. Es decir, tomará los lexemas, los agrupará según G3 y los evaluará para convertir el código en un flujo, no ya de palabras sin sentido, sino en un flujo de tokens (código tokenizado). La filmina ilustra en columnas la correspondencia entre lexemas y tokens, quedando en rojo el flujo de código tokenizado, que será el output o salida del scanner.

En síntesis estas filminas expresan que la función básica del scanner es reconocer una secuencia de lexemas y asociarlos a una secuencia de tokens. Para implementar este proceso el analizador léxico trabaja en base un conjunto de $AFD_{mínimos}$, construidos en base a las G3 que describen el léxico del lenguaje.

ANÁLISIS SINTÁCTICO

Si tenemos en cuenta que la sintaxis de los Lenguajes de Programación se especifica mediante gramáticas del tipo 2 en la jerarquía de Chomsky (G2 o independientes del contexto), entonces podemos decir que la función básica del ***parser*** o analizador sintáctico es utilizar los tokens suministrados por el scanner para reconocer frases o estructuras sintácticamente válidas en ***un lenguaje independiente del contexto***, produciendo como resultado final los árboles de análisis sintáctico.



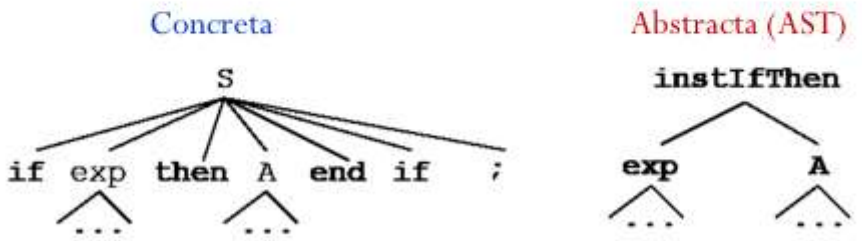
En este punto, cuando se habla de árboles de análisis sintáctico, se hace referencia a los árboles de **sintaxis concreta**, que son aquellos que se construyen en función de la gramática del lenguaje para una cadena concreta en la entrada. No abordaremos en este punto los árboles de sintaxis abstracta utilizados en fase de análisis semántico.

Análisis sintáctico

Árboles de sintaxis

- Concreta: sirve para el análisis sintáctico
- Abstracta: sirve para el análisis semántico

Sea: S → if *expr* then *A* endif ;



La pregunta que nos ocupará en adelante es ***¿Cómo se implementa este proceso?***

Análisis sintáctico

Analizadores recursivos

- cada no-terminal tiene asociada una rutina de análisis creada a partir de las reglas gramaticales
- scan de izquierda a derecha

Estrategias de análisis

- DESCENDENTE(Top-Down)
  - construye el árbol desde la raíz (S) hacia las hojas
  - problemas con recursividad a izquierda
- ASCENDENTE(Bottom-Up)
  - construye el árbol desde las hojas hacia la raíz (S)
  - se basa en encontrar una derivación por la derecha

Ya dijimos que un scanner (o analizador léxico) se puede implementar mediante un conjunto de Autómatas de Estados Finitos Deterministas ($AFD_{mínimo}$) que reconozca los tokens en un lenguaje regular (G3); y que un parser (o analizador sintáctico) se puede implementar mediante un Autómata de Pila que reconozca lenguajes independientes del contexto (G2).

Vimos que para los lenguajes regulares siempre hay un autómata determinista mínimo que permite un reconocimiento lineal[1]. Sin embargo para los lenguajes independientes del contexto no lo hay al menos que la gramática que lo genera (G2 o BNF) no sea ambigua y satisfaga también otras restricciones.

Para gramáticas BNF ***no ambiguas*** se han descubierto técnicas sencillas de análisis sintáctico que permiten construir parsers que reconocen en un tiempo lineal. Estos son los llamados ***Analizadores Recursivos***.

En particular, estas técnicas, proponen construir el árbol basándose en el reconocimiento de la cadena de entrada de ***izquierda a derecha***; y decidiendo en cada paso la regla gramatical que conviene elegir para continuar con la derivación. Los analizadores recursivos pueden clasificarse dependiendo de la forma en cómo se construyen los nodos del árbol de derivación sintáctico en:

- ✓ Los ***analizadores descendentes (Top-Down)*** que construyen el árbol desde la raíz hacia las hojas. Para ello procesan la entrada de izquierda a derecha y derivan por izquierda; por lo que también son llamados parsers ***LL (Left-to-right, Left-most derivation)***.

- ✓ Los ***analizadores ascendentes (Bottom-Up)***, que construyen el árbol desde las hojas hacia la raíz. Para ello procesan la entrada de izquierda a derecha y derivan por derecha; por lo que también son llamados LR ***(Left-to-right, Right-most derivation)***.

Veamos ahora en detalle como trabajan cada uno de estos tipos de parsers.

Parser LL(1) (Left-to-right, Left-most derivation) /Top-Down o Predictivos

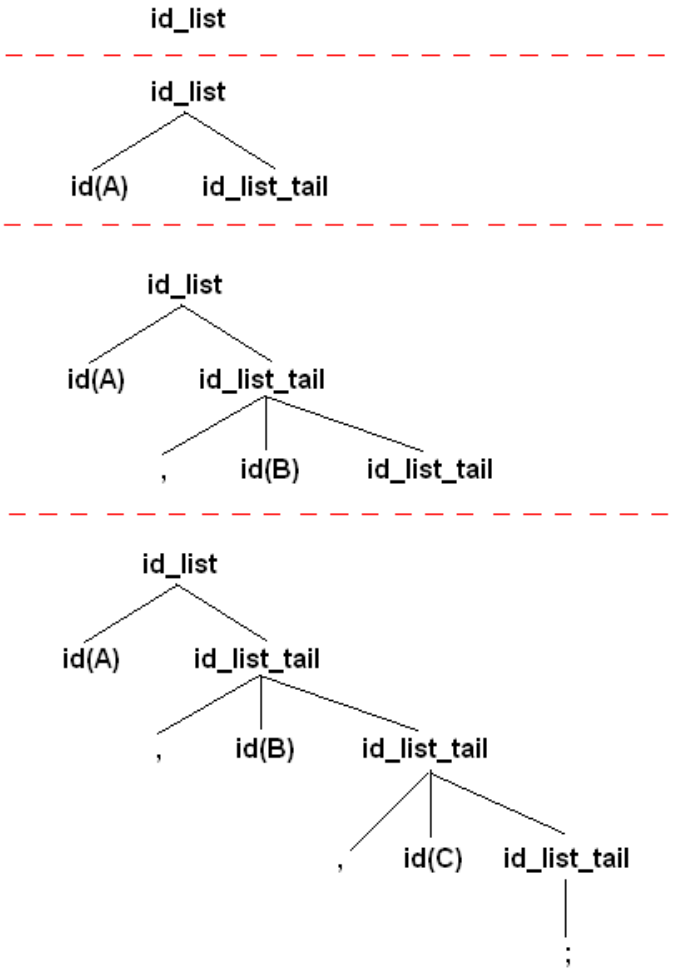
Construyen el árbol desde la raíz hacia las hojas, determinando en cada paso que producción es más conveniente para expandir el nodo actual, basándose en el próximo token disponible en la cadena de entrada, la cual recordemos barren de izquierda a derecha.

Ejemplo

Dado el siguiente conjunto de producciones que define la gramática para una lista de identificadores separados por coma y que termina en punto y coma; la figura muestra el proceso que haría un parser top-down (LL) para construir el árbol de derivación concreto para la cadena de entrada ***A,B,C;***

[1] Observar que el hecho de que el autómata sea determinista garantiza que el reconocedor ejecute el proceso en un tiempo lineal, es decir, en cada movimiento del autómata se reconoce un símbolo. Por lo tanto, el tiempo de ejecución de la máquina depende estrictamente de la longitud de la cadena a reconocer.

```
id_list      -> id id_list_tail
id_list_tail -> , id id_list_tail
id_list_tail-> ;
```



Como vemos el parser procesa la cadena de entrada de izquierda a derecha, y en cada paso, tomando en cuenta sólo el próximo token que le proporciona el scanner, decide que regla de producción aplicar para seguir derivando (observar que obviamente, esto implica derivar por izquierda).

Habitualmente, un parser recursivo descendente se implementa mediante un algoritmo que tiene:

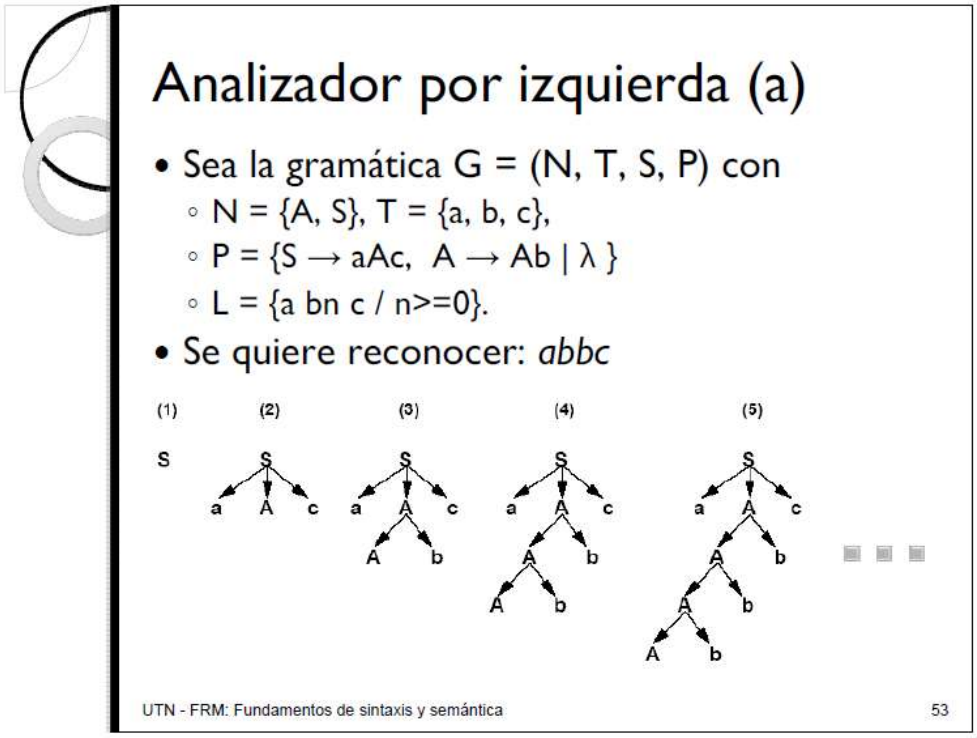
- ✓ Una subrutina para cada no terminal en la gramática.
- ✓ Un mecanismo para solicitar el input del próximo token proporcionado por el scanner
- ✓ Una rutina para consumir ese token en la medida que se encuentra una regla de producción con la que empata o matchea.

Error Sintáctico: Si el token proporcionado no es uno de los esperados de acuerdo a las reglas de producción de la gramática, la rutina de matcheo anunciará un ***error sintáctico.***

Si bien los analizadores recursivos top-down resultan, en general, más fáciles de comprender e implementar, no son aplicables a un gran número de gramáticas. Los principales obstáculos que encuentran los LL(1) son:

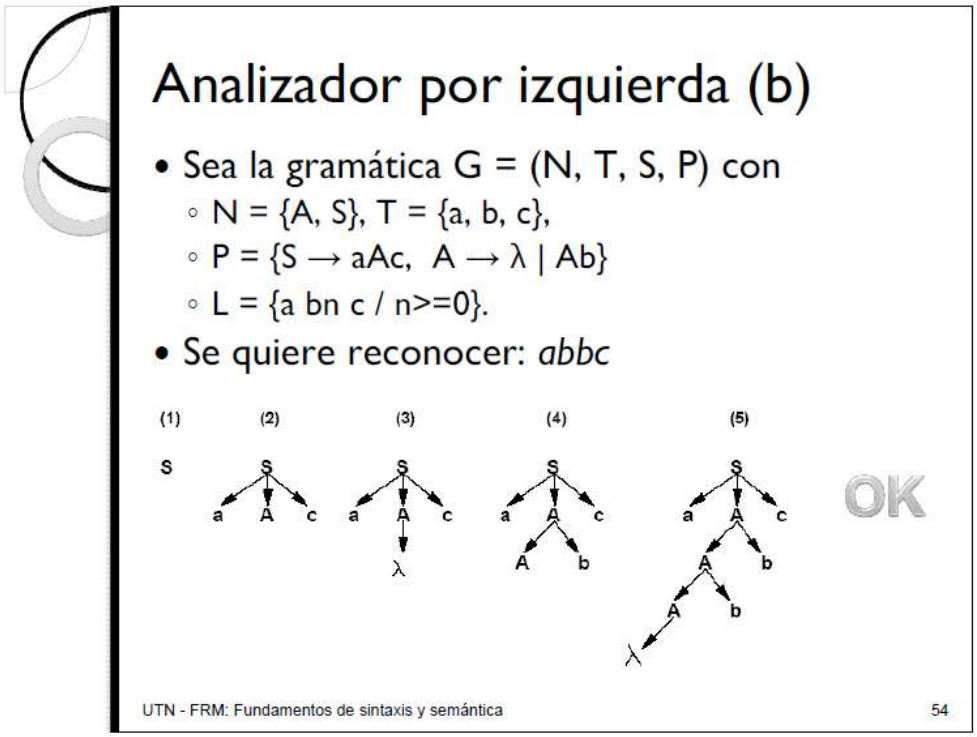
- Las producciones recursivas a izquierdas
- Los prefijos comunes.

La siguiente filmina ilustra el problema que presentan los parsers LL(1) con las ***gramáticas recursivas a izquierda.***



Como vemos, si bien la cadena pertenece al lenguaje descripto por la gramática, la estrategia LL(1) se traba en el paso (3), en el cuál el algoritmo que tiene como input el símbolo ***b***, no encuentra una regla de producción para el no terminal ***A*** que comience con ***b*** y por lo tanto no sabe cuál de las dos reglas escoger para seguir.

El problema anterior puede resolverse, como lo muestra la siguiente filmina, modificando el algoritmo para que pueda ***retroceder*** y probar otro camino hasta agotar todas las posibilidades antes de decidir rechazar una cadena. Sin embargo, este tipo de ***reconocedores top-down con retroceso*** serán, obviamente, más lentos y difíciles de programar que los reconocedores lineales.



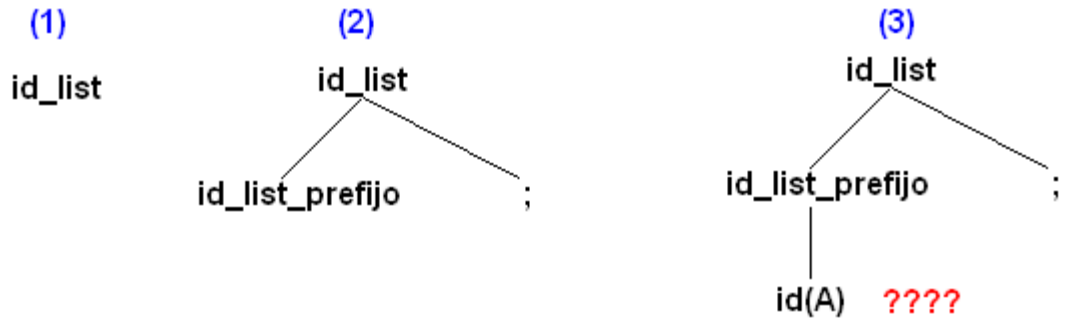
En definitiva, si la gramática presenta producciones recursivas a izquierda, un analizador Top-Down/LL(1) ***no*** puede procesarla, por lo que es deseable buscar una gramática equivalente sin recursividad a izquierda. (Recordar cómo eliminar la recursividad a izquierda).

Ejemplo

Una gramática equivalente, que es recursiva por izquierda, a la que vimos inicialmente para reconocer una lista de identificadores es:

```
id_list          -> id_list_prefijo ;
id_list_prefijo ->id_list_prefijo , id
                  | id
```

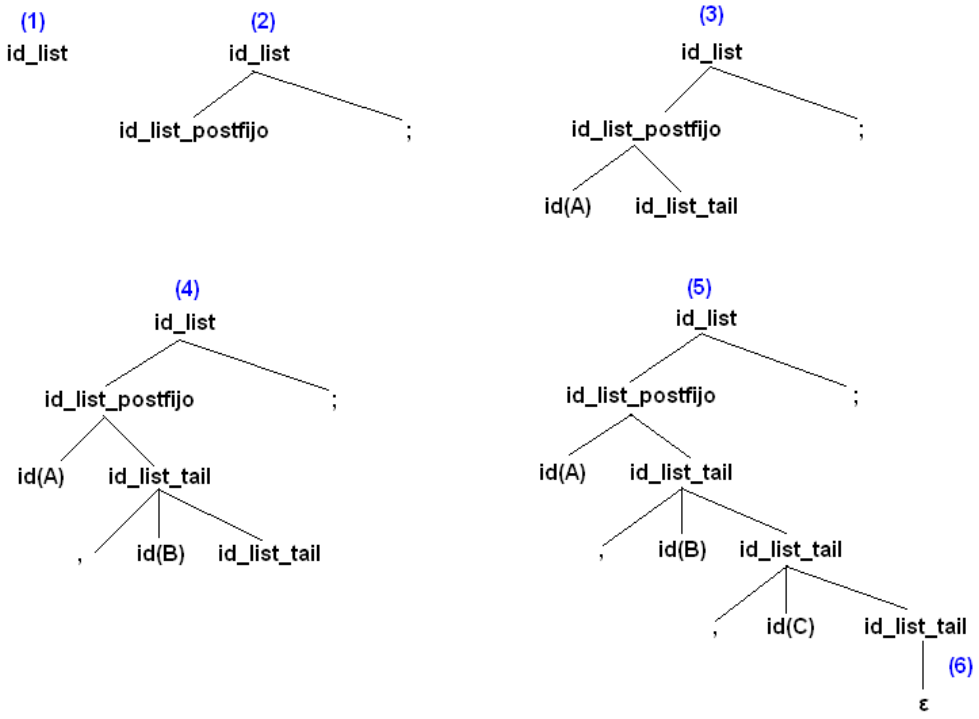
para la cadena A,B,C;



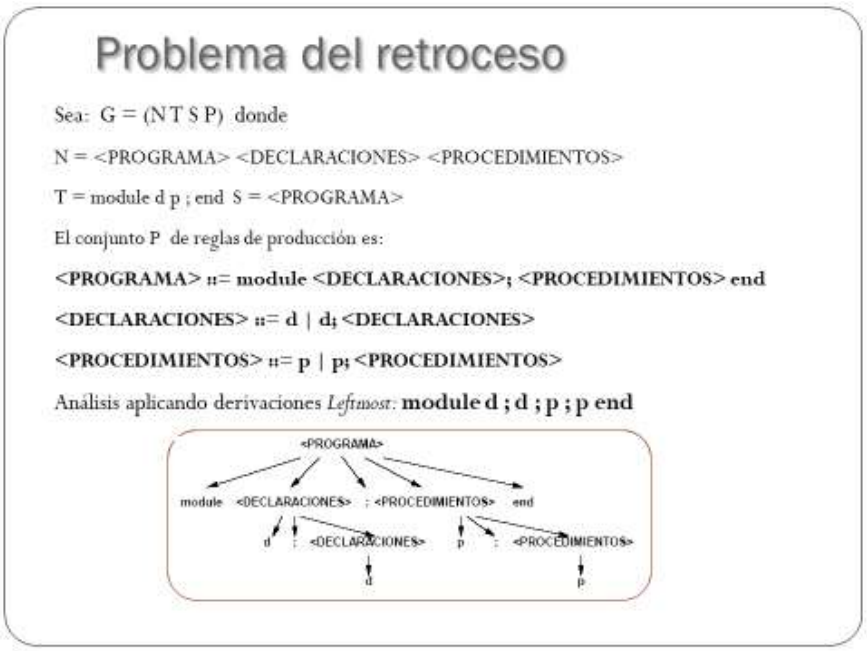
Busquemos ahora una gramática equivalente con el mecanismo descripto para eliminar la recursividad izquierda inmediata.

```
id_list          -> id_list_postfijo ;
id_list_postfijo -> id id_list_tail
id_list_tail     -> , id id_list_tail
                  | ε
```

para la cadena A,B,C;



El segundo tipo de obstáculo que enfrentan los analizadores recursivos descendentes, son las producciones con ***prefijos comunes,*** tal como ilustra el ejemplo de la siguiente filmina.

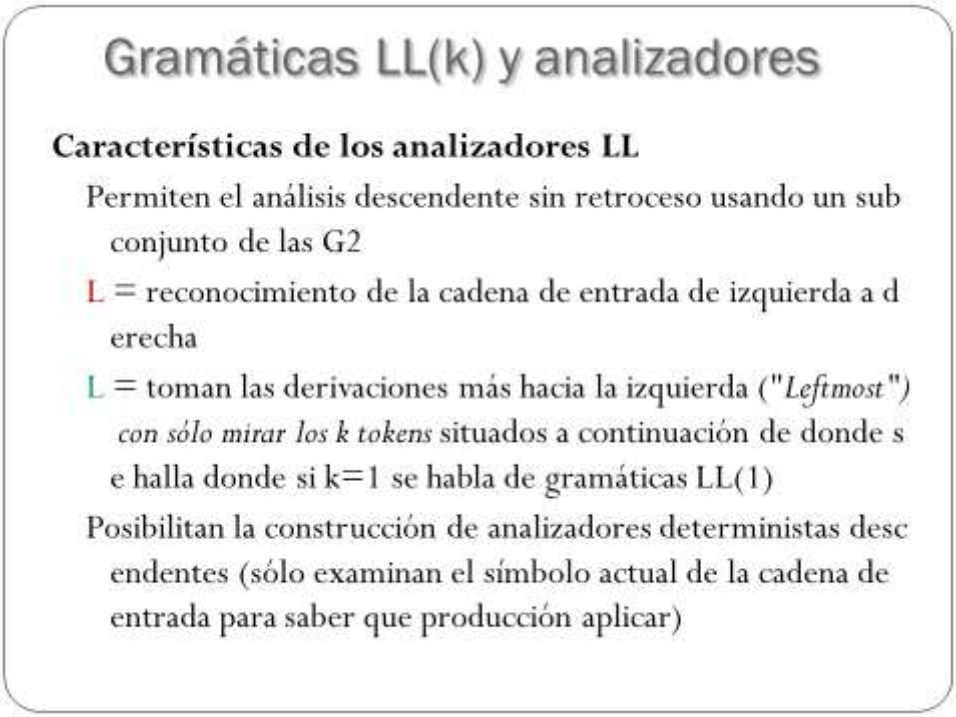


Como vemos, cuando en el input se proporciona el token ***d***; el analizador top-down no tiene forma de predecir que regla de producción aplicar, puesto que ***d,*** es un prefijo común de dos reglas de producción diferentes. Lo mismo sucede con el token ***b.***

Problema del Retroceso: Es claro que tanto la recursividad a izquierda como los prefijos comunes, conducen a la necesidad de implementar analizadores top-down con retroceso. Esto implica modificar el algoritmo para que se exploren todas las alternativas posibles cuando no se arriba a generar el árbol esperado. El retroceso además de complicar la programación del algoritmo, degrada significativamente su rendimiento.

Una solución alternativa y más eficiente a la implementación de reconocedores top-donw con retroceso para estos casos, son los traductores LL(k) que proponen un análisis anticipado de k tokens en la entrada antes de decidir que regla de producción aplicar.

- ✓ LL(1) traductores top-down con *un análisis anticipado de 1 caracter*
- ✓ LL(k) traductores "top-down" con Un análisis anticipado de k caracteres



Aun así, algunos problemas no se pueden resolver de forma descendente. Es decir, existen muchos lenguajes que ***no son LL(k***), para los cuáles no existe una gramática LL que los describa.

En estos casos se puede recurrir a las gramáticas LR, las cuáles están preparadas para ser reconocidas por analizadores ascendentes o botton-up.

Parser LR(1) (Left-to-right, Right-most derivation) o Bottom-Up/Shift-Reduce (Desplazamiento-Reducción)

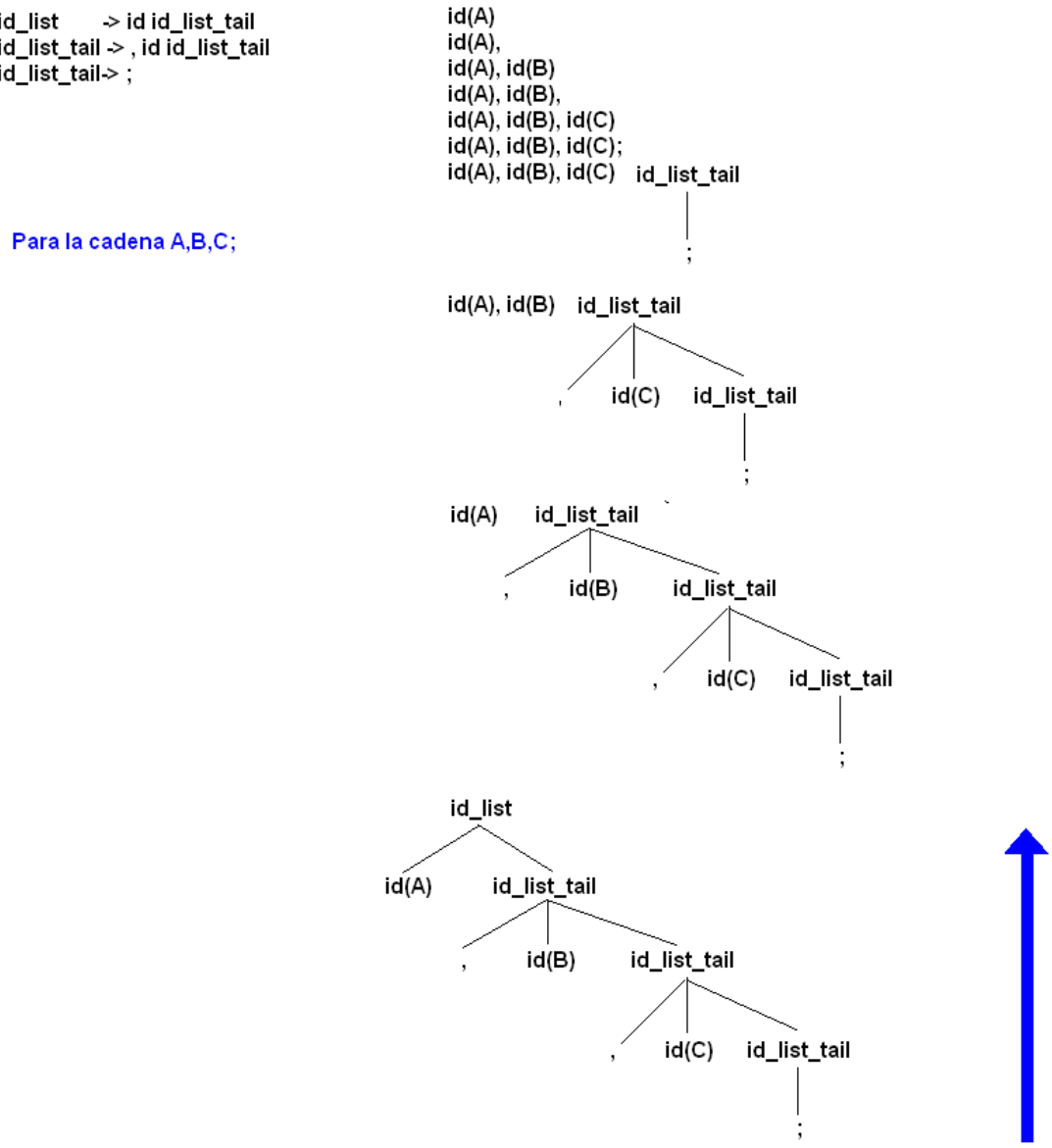
Construyen el árbol desde las hojas hacia la raíz, utilizando una técnica llamada ***desplazamiento-reducción (Shift-Reduce)*** que coincide con encontrar una derivación por derecha. Vamos a explicarlo con un ejemplo:

Ejemplo

Dado el siguiente conjunto de producciones que define la gramática para una lista de identificadores separados por coma y que termina en punto y coma; la figura muestra el proceso que haría un parser botton-up (LR(1)) para construir el árbol de derivación concreto para la cadena de entrada ***A,B,C;***

id_list -> id id_list_tail
id_list_tail -> , id id_list_tail
id_list_tail-> ;

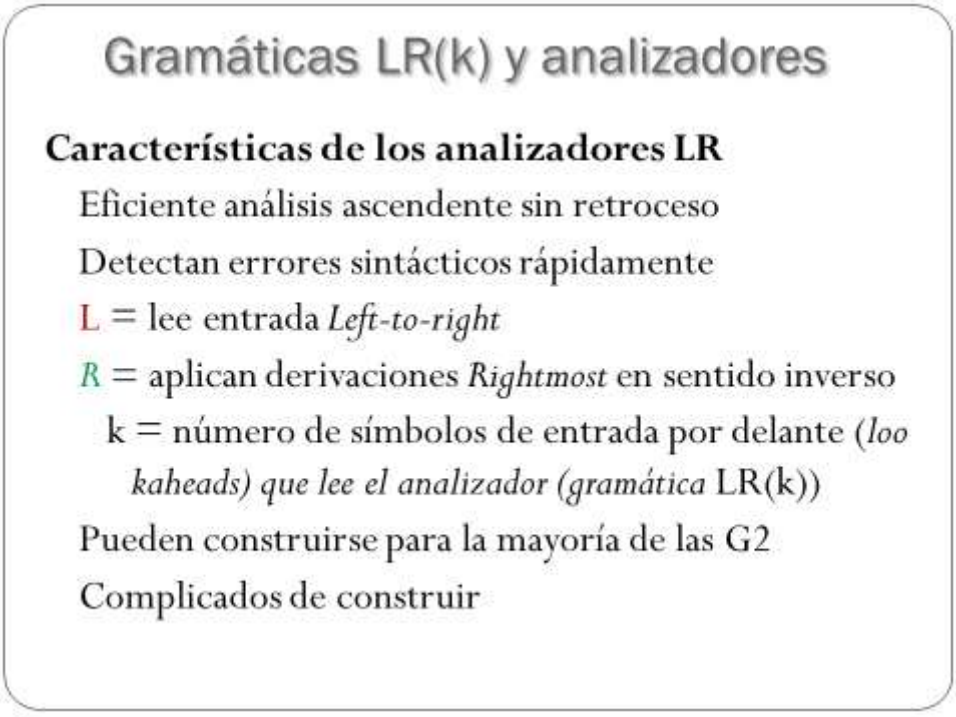
Para la cadena A,B,C;



El parser bottom-up comienza por tomar la hoja más a la izquierda del árbol (tal como lo lee en la cadena de entrada, de izquierda a derecha) y buscar si la puede reemplazar por la parte derecha de una regla de producción en la gramática. Como no puede, sigue leyendo y agrupando los símbolos en la cadena de entrada (hojas del árbol) hasta que puede reemplazarlos por el lado derecho de una regla de producción en la gramática. El proceso de agrupar un símbolo tras otro sin hacer nada más, se llama ***desplazamiento***, cuando se junta un grupo de símbolos que se puede sustituir por un no terminal a la derecha de una regla de producción se hace una ***reducción.***

En nuestro ejemplo, la primera reducción llega recién después de que el scanner proporciona el símbolo punto y coma. En este punto, se reduce el punto y coma por el no terminal id_list_tail, lo que permite a su vez reducir ahora la cadena ", id id_list_tail" a un nuevo id_list_tail. Así se sigue reduciendo y construyendo árboles parciales, hasta llegar al axioma de la gramática o raíz del árbol de derivación id_list.

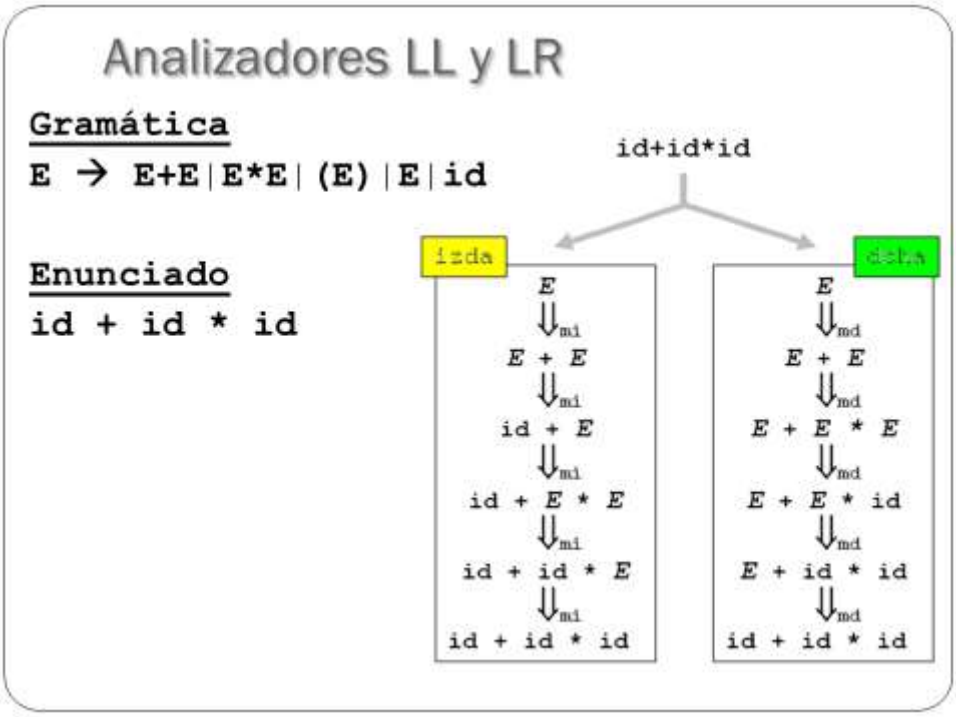
Si miramos el árbol desde abajo hacia arriba, veremos que corresponde al árbol de derivación por derecha.



En general se sostiene que además de poder generar la mayor parte de los lenguajes independientes del contexto, la estructura de las gramáticas LR resulta mucho más intuitiva que las LL, especialmente para aquellas gramáticas que describen expresiones aritméticas.

Las gramáticas LR en general se trabajan para que el parser sea construido por herramientas existentes tales como Bison o Yacc.

La siguiente filmina, ilustra comparativamente los procesos de derivación **MI** (más a la izquierda) y **MD** (más a la derecha) para expresiones aritméticas. Para comprenderla, debemos recordar que en ambos casos la cadena se lee siempre de izquierda a derecha. En MI, se deriva siempre el ***no terminal más a la izquierda***, mientras en que MD se deriva siempre el ***no terminal más a la derecha.***



Ahora vemos que la idea básica que subyace al comportamiento de los analizadores LR/botton-up o shift-reduce es:

> ***En cuanto hayan entrado suficientes token para detectar la parte derecha de una regla de producción, sustituirlos (reducirlos) por su parte izquierda.***

La siguientes filminas, que debe analizarse en su versión animada, ilustran el comportamiento de un analizador LR, el cuál utiliza una pila en la cual va desplazando los tokens que va proporcionando el scanner hasta que puede reducir (quitarlos de la pila) un grupo de ellos (en la cima de la pila) por un no terminal a la izquierda de una regla de producción.

Analizador ascendente

1 2 3 4 5

E → E + E | E * E | (E) | E | id

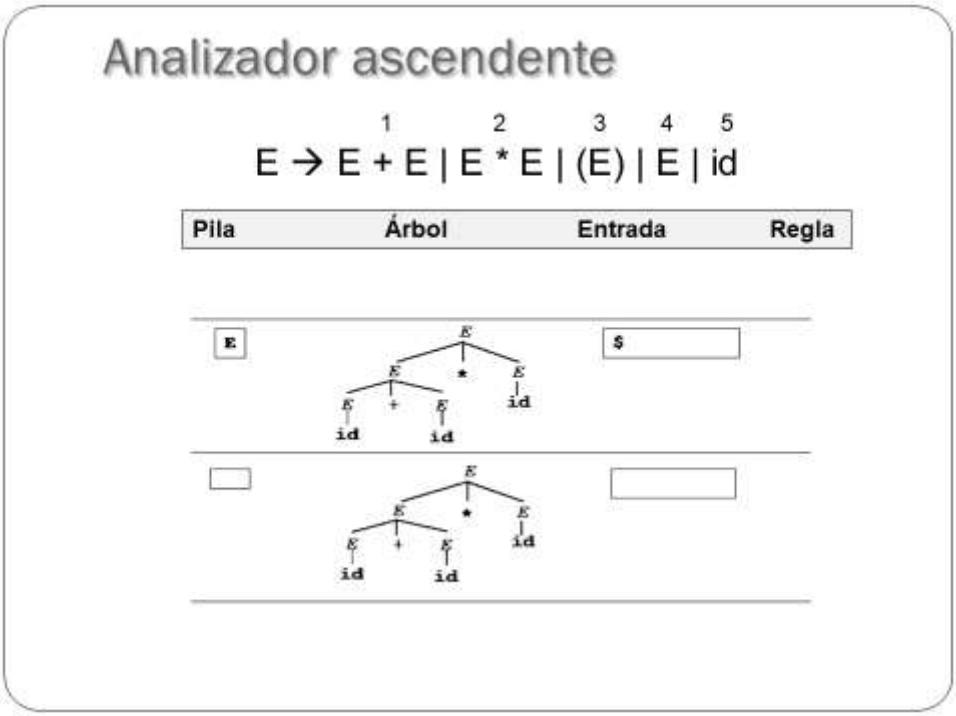
Pila	Árbol	Entrada	Regla
		id+id*id$	
id	id	+id*id$	5
E	E | id	+id*id$	
+ E	E + | id	id*id$	
id + E	E + | id id	*id$	5
E + E	E + E | | id id	*id$	1

Analizador ascendente

1 2 3 4 5

E → E + E | E * E | (E) | E | id

Pila	Árbol	Entrada	Regla
E	E(E + E) | | id id	*id$	
* E	E(E + E) * | | id id	id$	
id * E	E(E + E) * id | | id id	$	5
E * E	E(E + E) * E(id) | | id id	$	2



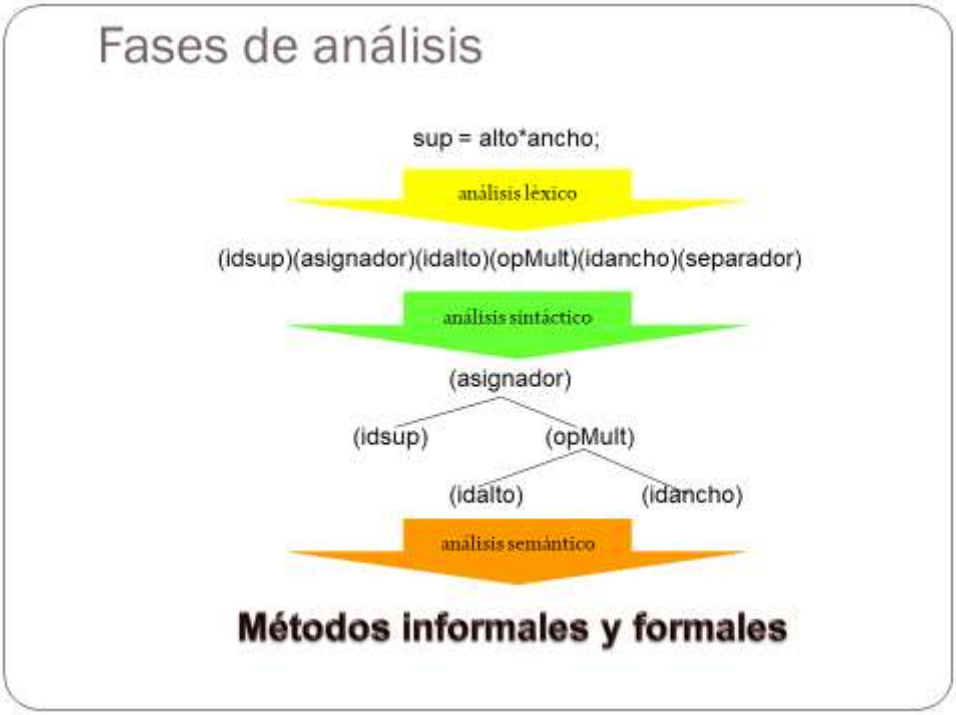
Tanto los parser LL como los LR son utilizados en compiladores en producción. No obstante suelen ser más comunes los LR, entre otras cosas porque existen herramientas útiles para diseñarlos a partir de la descripción de la gramática, que resuelven solas su implementación tales como Yacc o Bison.

SEMÁNTICA DE LENGUAJES

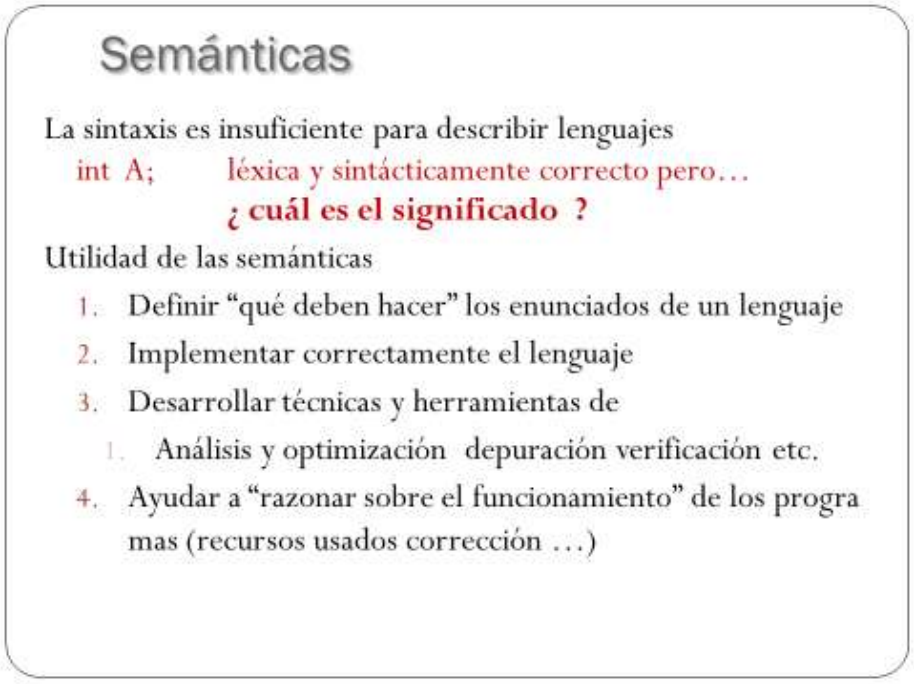
Hasta ahora hemos visto como describir el léxico y la sintaxis de un lenguaje. Es decir, la forma o estructura de las palabras, expresiones, enunciados y unidades de un programa.

Sin embargo, se necesita más que la definición de las estructuras léxicas y sintácticas para describir completamente un lenguaje.

El compilador o traductor, debe poder realizar una interpretación no ambigua de los enunciados, declaraciones, expresiones, etc., para poder entonces producir el código objeto del programa. La Semántica define el significado de las expresiones, enunciados y unidades de un programa.



El problema de la definición semántica de lenguajes ha sido objeto de estudio durante tanto tiempo como el problema de la definición sintáctica; pero en general, no se han encontrado soluciones eficientes y útiles a la Teoría de Compiladores que permitan lograr una definición ***"formal"*** (en términos matemáticos) única y adecuada de la semántica de un lenguaje.

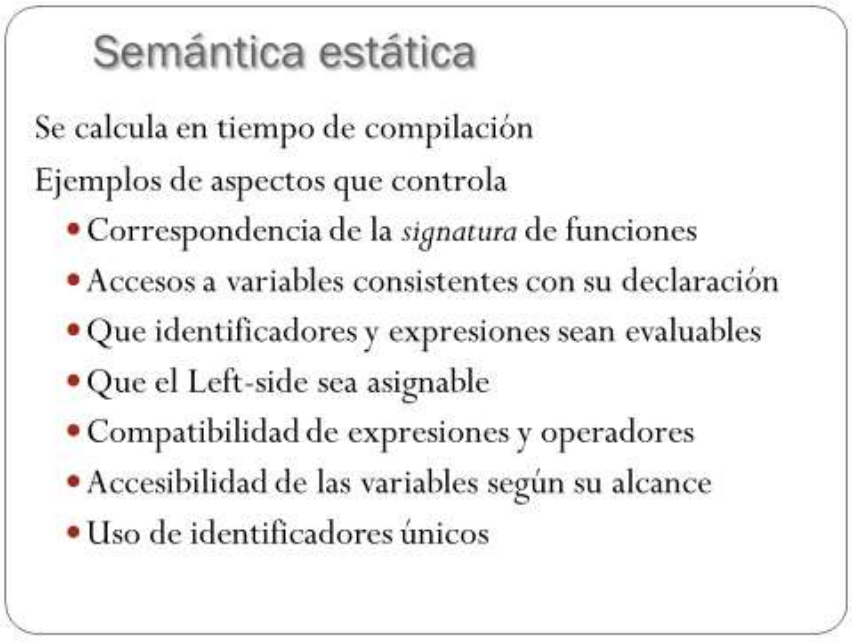


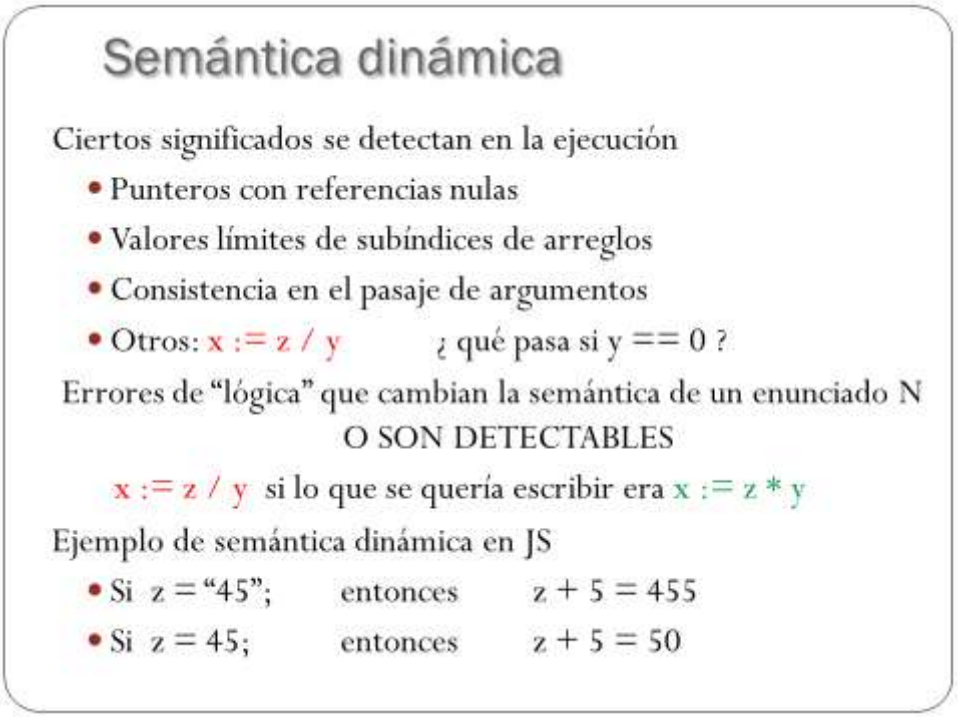
La definición semántica en un lenguaje describe "cómo se implementan" (cómo se resuelven) diversos aspectos del lenguaje. Por ejemplo: el formato de representación en memoria, a nivel de hardware de los números enteros, o el tipo de argumentos que se puede pasar a un operador.

Dependiendo del momento en que se resuelven estos aspectos, la implementación de la semántica puede ser de dos tipos.

- ***Semántica Estática:*** reúne a todos aquellos aspectos semánticos de un lenguaje que se calculan, se resuelven o se deciden en tiempo de compilación.
- ***Semántica Dinámica:*** reúne a todos aquellos aspectos semánticos de un lenguaje que se calculan, se resuelven o se deciden en tiempo de ejecución.

Las siguientes filminas ilustran algunos ejemplos:





Al diseñar un lenguaje de programación se debe decidir que aspectos semánticos se implementan de manera estática y cuáles de forma dinámica. Es decir, no necesariamente un lenguaje tiene semántica estática o semántica dinámica, sino que en general conviven los dos tipos de definiciones semánticas en la implementación de un lenguaje, para aspectos diferentes.

Según el diseño de un Lenguaje en particular, algunos de los ejemplos vistos podrían decidirse también de manera dinámica (por ej.: verificación de tipos en lenguajes débilmente tipeados).

En la segunda unidad de la asignatura, veremos con mayor detalle las ventajas y desventajas de una u otra forma de implementación de la semántica de un lenguaje, para cada uno de los aspectos involucrados en su estructura.

Sea cual sea la forma en que se especifique la semántica de un lenguaje, ya sea por métodos informales o formales, es deseable que dicha especificación cumpla las características enumeradas en la siguiente filmina.

En general, la semántica de los lenguajes se especifica de manera ***"no formal"*** o ***"coloquial"***, es decir, en prosa ordinaria. Típicamente, como se puede observar en los Manuales de Programación, se da una regla (o conjunto de reglas) BNF u otra gramática formal que define la sintaxis de una construcción, y luego se proporcionan unos cuántos párrafos y ejemplos para definir la semántica. El problema con esto es que no garantiza que los lectores hagan la interpretación correcta de lo que está escrito. La prosa es ambigua. Esto hace que estos lenguajes no sean totalmente confiables.

No obstante lo anterior, existen algunos modelos para definir Semánticas Formales, que aunque no han tenido gran aplicación en la práctica por ser muy complejos, debemos mencionar brevemente. Su estudio en detalle corresponde a un estudio más profundo de la Teoría de Compiladores, que excede el alcance de este curso.

La siguiente filmina, enumera algunas de estas técnicas de especificación de semántica y las evalúa respecto de la medida en que cumplen con las características deseadas.

S – si cumple
R – cumple moderadamente

Técnicas de especificación semántica

Semántica	No ambigua	Modular	reusable	demostrar	prototipar	legible	flexible	experiencia
Lenguaje natural		S	S			R	S	S
Operacional	S			R	R	R	S	S
Denotacional	S			S	R		S	R
Axiomática	R			S		R		S
Algebráica	S		R	S	S	R	R	R
De estado abstacto	S	S	R	R	S		R	S
De acción	S	S	R	R	S		R	R
Monádica modular	S	S		S	R		S	
Monádica reutilizable	S	S	S	S	S		S	

Para finalizar esta sección de Etapas de Análisis, evaluemos con casos prácticos los distintos tipos de errores que pueden presentarse en un programa.

Tipos de Errores

SINTAXIS

```
int main ()
   int pt qr*;
   system("pause");


int main ()
   int pt qr
   !qr;
  system("pause");


int main ()
   int pt qr;
   ¡qr¡
   system("pause");
```

SEMÁNTICA ESTÁTICA

```
int main ()
   int pt = 1;
   if (pt != 0)
        int qr;
        qr = pt;

   cout << qr << endl;
   system("pause");


int main ()
   long pt = 9687372626;
   int qr = 1425;
   pt = qr;
   cout << pt << endl;
   system("pause");
```

SEMÁNTICA DINÁMICA

```
int main ()
   int pt = 1 qr;
   if (pt != 0)
        int qr = 3;
        qr = pt;

   cout << qr << endl;
   system("pause");


int main ()
   long pt = 68737262;
   int qr = 7425;
   pt = qr * pt;
   cout << pt << endl;
   system("pause");
```