PROCESADORES DE LENGUAJE

Tema 2: Las fases de análisis del compilador

Departamento de Ingeniería Informática
Universidad Francisco de Vitoria

Curso 2012/13

Objetivos del tema

UNIVERSIDAD FRANCISCO DE VITORIA

- Objetivos
  - Entender el procesamiento de entrada de caracteres durante el análisis léxico.
  - Definir notación para la especificación de expresiones regulares.
  - Refrescar el conocimiento sobre autómatas finitos deterministas (AFD) y no deterministas (AFN) , aplicado al reconocimiento de patrones de gramáticas G-3.
  - Algoritmo de traducción de AFN a AFD.
  - Algoritmo para la definición de AFN reconocedores de una expresión regular dada.
  - Definición de gramáticas independientes de contexto.
  - Conocer la estructura y notaciones de una gramática.
  - Detalles sobre análisis sintáctico descendente y ascendente.
  - Entender gramáticas LL(1)

Contenido

UNIVERSIDAD FRANCISCO DE VITORIA

- Tratamiento de la entrada
- Gramáticas regulares y reconocedores: autómatas finitos.
- Gramáticas independientes de contexto y analizadores sintácticos
- Gramáticas LL(k) y analizadores descendentes

Bibliografía recomendada

UNIVERSIDAD FRANCISCO DE VITORIA

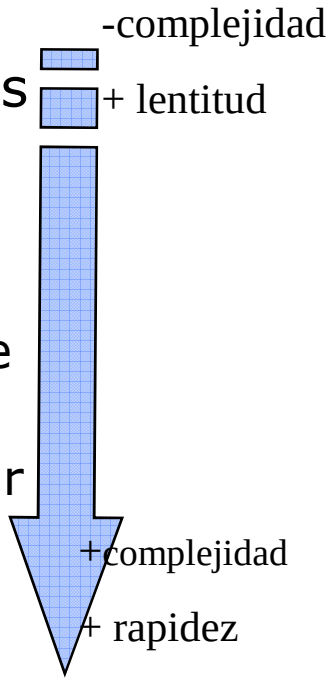
- ***Compiladores*** (Principios, técnicas y herramientas)
  PEARSON, Addison Wesley Longman
  - Capítulo 3: Analizador léxico
  - Capítulo 4: Analizador sintáctico

1. Tratamiento de la entrada

UNIVERSIDAD FRANCISCO DE VITORIA

Varios métodos para la implementación de analizadores léxicos:

- Utilizar generador de analizador léxico (LEX)
- Escribir analizador en lenguaje convencional o utilizar capacidad de dicho lenguaje para el reconocimiento de expresiones, dado un patrón regular (Perl)
- Programar analizador léxico en ensamblador y manejar directamente la lectura de la entrada



- El consumo de tiempo requerido por esta etapa normalmente es significativo, pese a ser una labor conceptualmente sencilla.
  - Se elaboran algunas técnicas con el objetivo de mejorar el rendimiento.

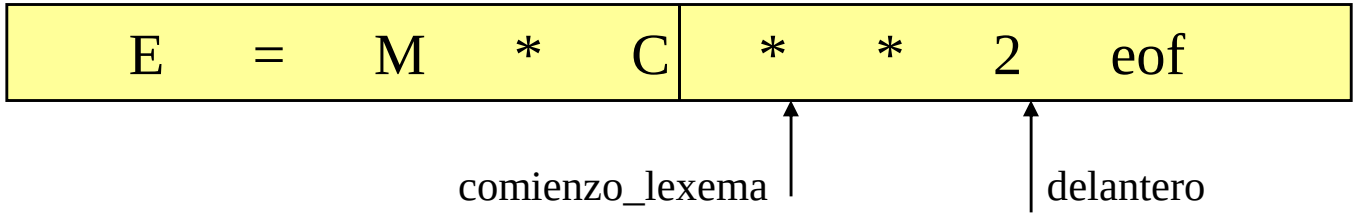
1. Tratamiento de la entrada

UNIVERSIDAD FRANCISCO DE VITORIA

Parejas de Buffers

- En muchos lenguajes es necesario preanalizar varios caracteres, además del lexema de un patrón, antes de poder anunciar su concordancia.
- Las técnicas basadas en *buffers* están especializadas en reducir el número de operaciones necesarias.
- Se utiliza un *buffer* dividido en dos mitades de *N* caracteres cada uno (normalmente un múltiplo del tamaño de bloque de disco, 1024, 4096).

1. Tratamiento de la entrada



- Se mantienen 2 apuntadores de entrada. Uno siempre permanece apuntando al primer carácter del lexema que se analiza y el segundo recorre los caracteres posteriores hasta encontrar concordancia con algún patrón.

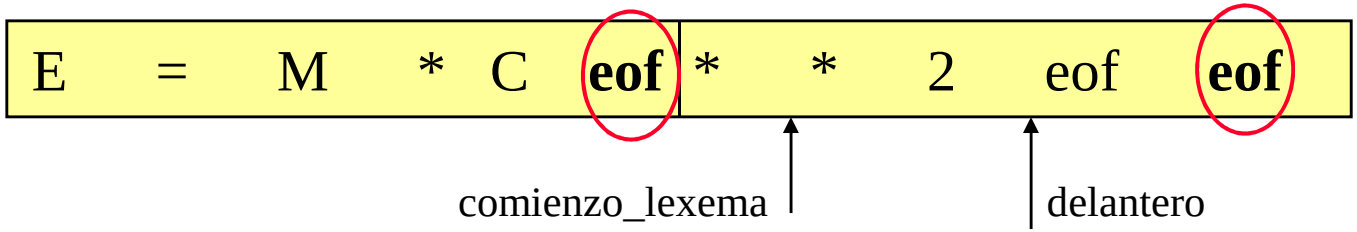
- Cuando el reconocimiento de un lexema se ha completado, ambos apuntadores pasan al siguiente carácter del último usado.

- Cuando el puntero delantero se aproxima al final de la primera mitad del buffer, se rellena la mitad de la derecha con N nuevos caracteres. Se realiza el proceso inverso cuando se agota la mitad derecha.

1. Tratamiento de la entrada

UNIVERSIDAD FRANCISCO DE VITORIA

- Esta técnica permite la eliminación de *blancos* y *comentarios* en tiempo de análisis léxico.



Centinelas

- También se puede añadir una carácter especial al final de cada *buffer*, no perteneciente a la entrada de código, con el propósito de identificar el final de *buffer*, y poder comenzar a rellenar la otra mitad.

2. Gramáticas regulares y reconocedores: autómatas finitos

UNIVERSIDAD FRANCISCO DE VITORIA

Definiciones

- **Componente léxico:** Elemento más simple con significado propio de un lenguaje de programación.
- Ejemplos:
  - Identificadores: variables, palabras claves, ...
  - Números
  - Cadenas de caracteres
  - Operadores: aritméticos, relacionales, lógicos, ...
  - Signos de puntuación
- También se denominan "tokens"
- Son los **símbolos terminales** de las gramáticas de contexto libre que generan los lenguajes de programación.

2. Gramáticas regulares y reconocedores: autómatas finitos

UNIVERSIDAD FRANCISCO DE VITORIA

- **Tabla de Símbolos:** Se crea durante el Análisis Léxico. Puede almacenar:
  - Números
  - Cadenas
  - ...
  - Pero, sobre todo, identificadores
- Cuando el analizador léxico reconoce un identificador:
  - Se inserta el identificador en la Tabla de Símbolos
  - Devuelve el componente léxico de identificador y un puntero o índice a su posición en la Tabla de Símbolos.
- La información del identificador depende de su naturaleza:
  - Variable: tipo de dato, valor, ...
  - Función: número y tipo de argumentos, ...
  - Etc.

2. Gramáticas regulares y reconocedores: autómatas finitos

UNIVERSIDAD FRANCISCO DE VITORIA

- Ejemplo (dividendo = divisor * cociente + resto)

Nombre	Tipo	Valor	...
dividendo			
divisor			
cociente			
resto			

- Nota: La información de los identificadores se completa durante todas las fases del proceso de traducción.

2. Gramáticas regulares y reconocedores: autómatas finitos

UNIVERSIDAD FRANCISCO DE VITORIA

- **Palabras claves o reservadas:**
  - Son identificadores con un significado especial
  - Algunos lenguajes de programación también permiten usar las palabras claves como variables
    - Las palabras claves no serían palabras reservadas
    - Dificulta la legibilidad de los programas

- Ejemplo: **IF** (VAR2 > 1) **THEN** VAR1 = 0;

- Tipos de reconocimiento de las palabras claves:
  - Implícito: preinstalación en la Tabla de Símbolos
  - Explícito: reconocimiento específico de cada palabra clave.

2. Gramáticas regulares y reconocedores: autómatas finitos

UNIVERSIDAD FRANCISCO DE VITORIA

Definiciones

- El **alfabeto** denota cualquier conjunto finito de símbolos. Ejemplos típicos son las letras y caracteres. El conjunto {0,1} compone el alfabeto binario.

- Una **cadena** sobre un alfabeto es una secuencia finita de símbolos tomados de ese alfabeto. También se utiliza a veces *palabra o frase* para referirnos a cadena de caracteres. La longitud de la cadena *s* se puede escribir como |*s*|. La cadena vacía existe y se representará como "*ε*" *(o "λ").*
- .

2. Gramáticas regulares y reconocedores: autómatas finitos

UNIVERSIDAD FRANCISCO DE VITORIA

Operaciones aplicadas al lenguaje

- Definición: L es un lenguaje formal definido sobre Σ si $L \subseteq \Sigma^*$

- Hay varias operaciones importantes que se pueden aplicar a los lenguajes. Una operación , aplicada sobre 2 o más lenguajes, produce un nuevo lenguaje. Para el análisis léxico, interesan principalmente la unión, la concatenación y la cerradura (potencia).

- Ejemplo: Sea *L* el conjunto {A, B, ...Z, a, b, ... z} y *D* {0, 1, ... 9},
  - *L* ***U*** *D* es el conjunto de caracteres y dígitos.
  - *LD* es el conjunto de cadenas que consta de una letra seguida de un dígito
  - L^4 es el conjunto de todas las palabras que tienen 4 letras.
  - L^* es el conjunto de todas las cadenas de letras, incluyendo la cadena vacía.
  - L^+ es $L^* - \varepsilon$

UNIVERSIDAD FRANCISCO DE VITORIA

Expresiones regulares

- En pascal, un identificador es una letra seguida de 0 o más letras o dígitos. Con esta notación, se pueden definir los identificadores de Pascal como:
  - *letra (letra | dígito)**
- *Identificador en lenguaje C:*
  - *(subrayado | letra) (subrayado | letra | cifra)**
- Las siguientes son las reglas que definen las *expresiones regulares del alfabeto* Σ,
  1. ε es una expresión regular designada por $\{\varepsilon\}$.
  2. Si *a* es un símbolo de Σ entonces *a* es una expresión regular designada por $\{a\}$

2. Gramáticas regulares y reconocedores: autómatas finitos

UNIVERSIDAD FRANCISCO DE VITORIA

1. Suponiendo que *r* y *s* sean expresiones regulares representadas por los lenguajes *L(r)* y *L(s)*.
   - (r)|(s) es una expresión regular representada por *L(r)* **U** *L(s)*
   - (r)(s) es una expresión regular representada por *L(r)L(s)*
   - (r)* es una expresión regular representada por *L(r)**
   - (r) es una expresión regular representada por *L(r)*.
   - $(r)^+$ es una expresión regular representada por *L(r)** - ε

- Prioridad de los operadores de las expresiones regulares:

  + Máxima prioridad ( )

  *

  .

  - Mínima prioridad |

2. Gramáticas regulares y reconocedores: autómatas finitos

UNIVERSIDAD FRANCISCO DE VITORIA

Definiciones regulares

- Por conveniencia de notación, puede ser deseable dar nombres a las expresiones regulares y definir expresiones regulares utilizando dichos nombres como si fueran símbolos.

$$
\begin{aligned}
d1 &\rightarrow r1 \\
d2 &\rightarrow r2 \\
&\ldots \\
dn &\rightarrow rn
\end{aligned}
$$

2. Gramáticas regulares y reconocedores: autómatas finitos

UNIVERSIDAD FRANCISCO DE VITORIA

- Ejemplo 1:

letra $\rightarrow$ *A | B | ... | Z | a | b | ... | z*
dígito $\rightarrow$ *0 | 1 | ... | 9*
id $\rightarrow$ ***letra (letra|digito)****

- Ejemplo 2:

dígito $\rightarrow$ *0 | 1 | ... | 9*
dígitos $\rightarrow$ ***digito***$^{+}$
fraccion_operativa $\rightarrow$ *(.* ***digitos*** *)|* ε
expo_operativo $\rightarrow$ *(E (+|-|* ε*)* ***digitos*** *)* | ε
num $\rightarrow$ ***dígitos | fraccion_operativa | expo_operativo***

2. Gramáticas regulares y reconocedores: autómatas finitos

UNIVERSIDAD FRANCISCO DE VITORIA

- Ejemplo 3: Números en el Lenguaje C

numero $\rightarrow$ parte_entera (ε | parte_decimal)

parte_entera $\rightarrow$ cifra cifra*

parte_decimal $\rightarrow$

punto cifra*(ε | (E + e)(ε | ″ − ″ | ″ + ″)cifra cifra*)

2. Gramáticas regulares y reconocedores: autómatas finitos

UNIVERSIDAD FRANCISCO DE VITORIA

Autómatas finitos

- Un *reconocedor* de un lenguaje es un programa que toma como entrada una cadena *x* y responde *"sí"* si *x* es una frase del programa, y *"no"*, si no lo es. Se compila una expresión regular en un reconocedor construyendo un diagrama de transiciones generalizado llamado autómata finito.

- Un autómata finito puede ser determinista (AFD) o no determinista (AFN)

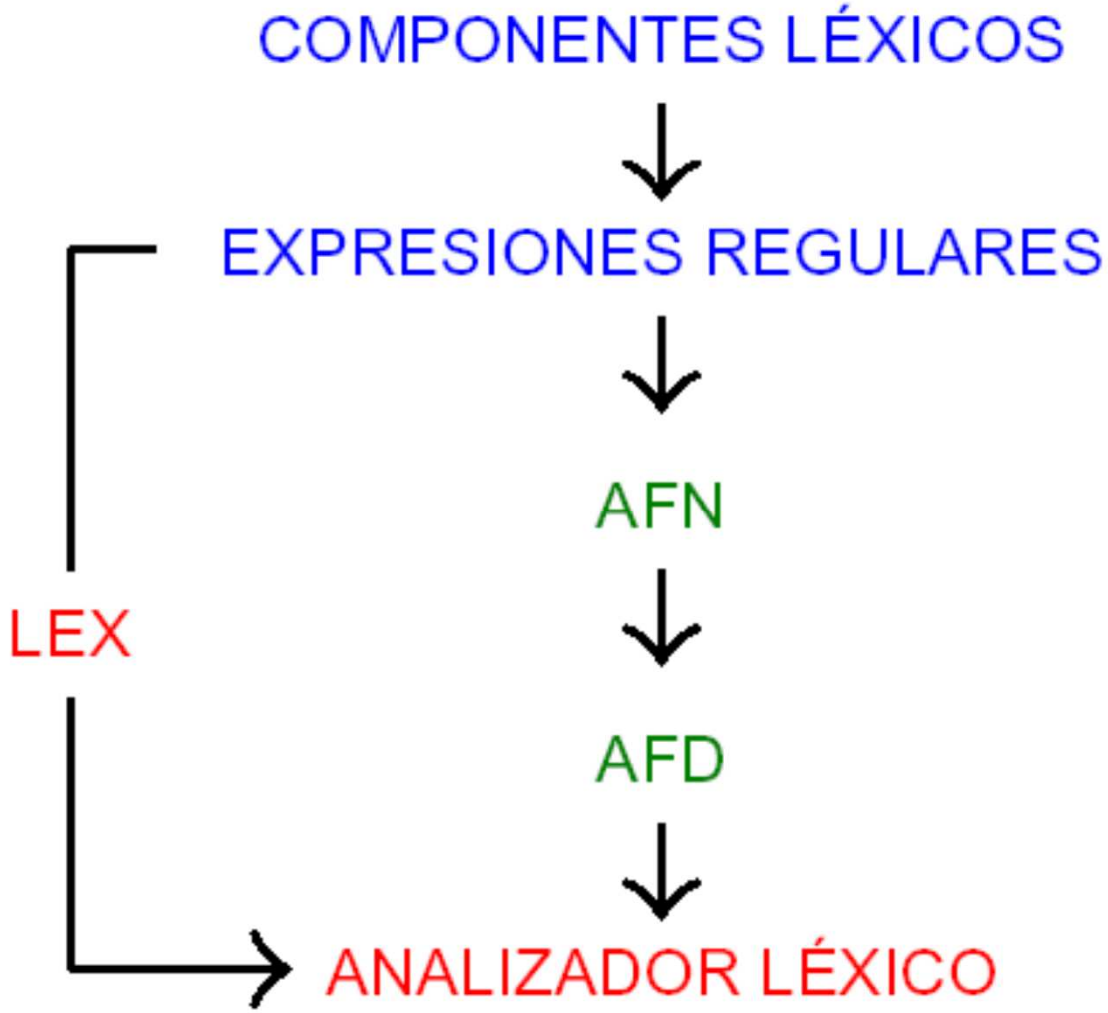
- El autómata no determinista tiene varias transiciones de estado para un mismo símbolo de entrada desde un estado determinado. El determinista solo tiene una única transición por estado y símbolo de entrada.

2. Gramáticas regulares y reconocedores: autómatas finitos

UNIVERSIDAD FRANCISCO DE VITORIA

- Tanto AFN como AFD son capaces de reconocer a los conjuntos regulares.
- Hay un conflicto entre espacio y tiempo. AFN usan poco espacio pero son más lentos. Situación inversa en AFD.
- Es posible convertir cualquier AFN en AFD mediante una procedimiento sistemático.

2. Gramáticas regulares y reconocedores: autómatas finitos



UNIVERSIDAD FRANCISCO DE VITORIA

Autómatas finitos no deterministas

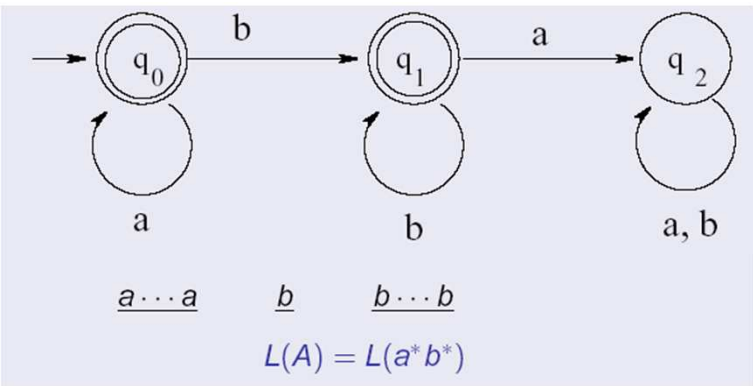
- Un AFN es un modelo matemático definido por:
  - Un conjunto de estados *S*
  - Un conjunto de símbolos de entrada Σ
  - Una función de transición *mueve* que transforma los pares *estado-símbolo* en conjuntos de estados.
  - Un estado s*0* que se considera el estado de inicio
  - Un conjunto de estados *F* considerados como *estados de aceptación*.

- Ejemplo: Grafo y tabla de transiciones para los lenguajes (a|b)*abb y aa*|bb*

Autómatas finitos deterministas (AFD)

- Un AFD es un caso especial de AFN donde se cumplen las siguientes condiciones:
  - ningún estado tiene transición vacía.
  - para cada estado *s* y símbolo de entrada *a*, hay a lo sumo una arista etiquetada *a* que sale de *s*.

- Ejemplo: Grafo y tabla de transiciones para el lenguaje a*b*.



$$
\begin{aligned}
(q_0, \underline{a}abb) &\vdash (q_0, \underline{a}bb) \\
&= (q_0, \underline{b}, b) \\
&= (q_1, \underline{b}) \\
&= (q_1, \epsilon)
\end{aligned}
$$

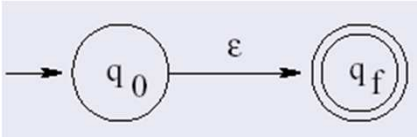
Construcción de un AFN para una expresión regular

- *Algoritmo de Thompson: genera un AFN a partir de una expresión regular.*

$$L((\alpha)) = L(\alpha) = L(A_N)$$

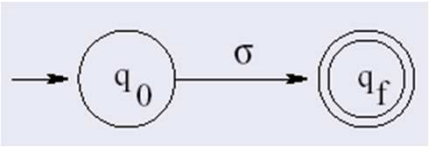
  - *AFN equivalente a la expresión regular ∅:*

  - *AFN equivalente a la expresión regular ε:*

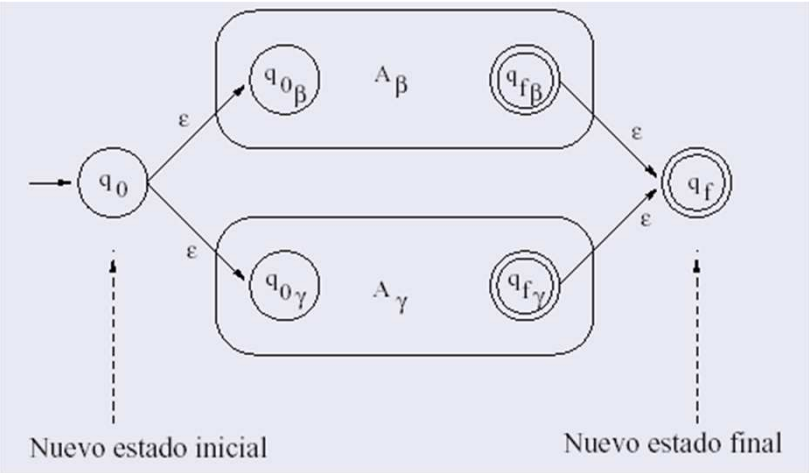


2. Gramáticas regulares y reconocedores: autómatas finitos

- *AFN equivalente a la expresión regular σ:*

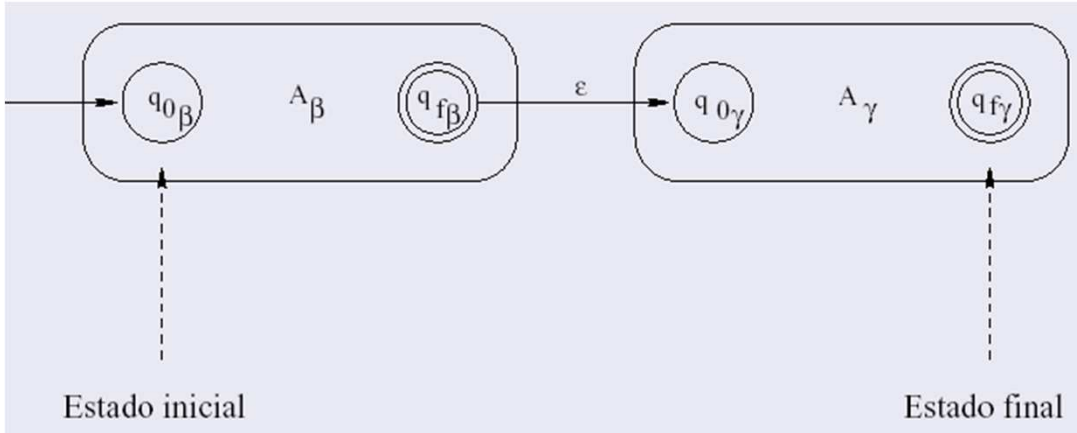


- *AFN equivalente a la expresión regular α | β:*

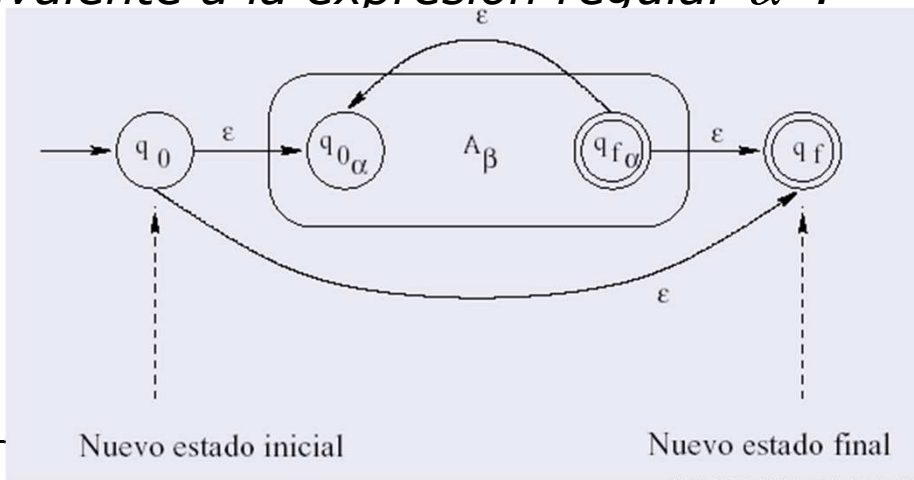


2. Gramáticas regulares y reconocedores: autómatas finitos

- *AFN equivalente a la expresión regular $\alpha\beta$:*



- *AFN equivalente a la expresión regular α^*:*



2. Gramáticas regulares y reconocedores: autómatas finitos

UNIVERSIDAD FRANCISCO DE VITORIA

- Ejercicio 1: Definir AFN de expresión $\alpha = l\ (l \mid d)^*$
- Ejercicio 2: Definir AFN de la expresión (a|b)*abb
- Ejercicio 3: Definir AFN de la expresión:
  - numero $\rightarrow$ parte_entera (ε | parte_decimal)
  - parte_entera $\rightarrow$ cifra cifra*
  - parte_decimal $\rightarrow$ punto cifra cifra*(ε | E(”−” | ”+”)cifra cifra*)

  considerar cifra como elemento terminal representado por *“a”* y *punto* representado por *“b”, E como “c”, - como “d” y + como “e”.*

UNIVERSIDAD FRANCISCO DE VITORIA

Conversión de un AFN en un AFD

- Es posible convertir un AFN en AFD mediante el algoritmo de *construcción de subconjuntos.*
  - ***cerradura-e(s):*** conjunto de estados del AFN alcanzables desde el estado s del AFN con transiciones *vacías* solamente.
  - ***mueve(T,a):*** conjunto de estados del AFN hacia los cuales hay una transición con el símbolo de entrada a desde algún estado s en T del AFN.

2. Gramáticas regulares y reconocedores: autómatas finitos

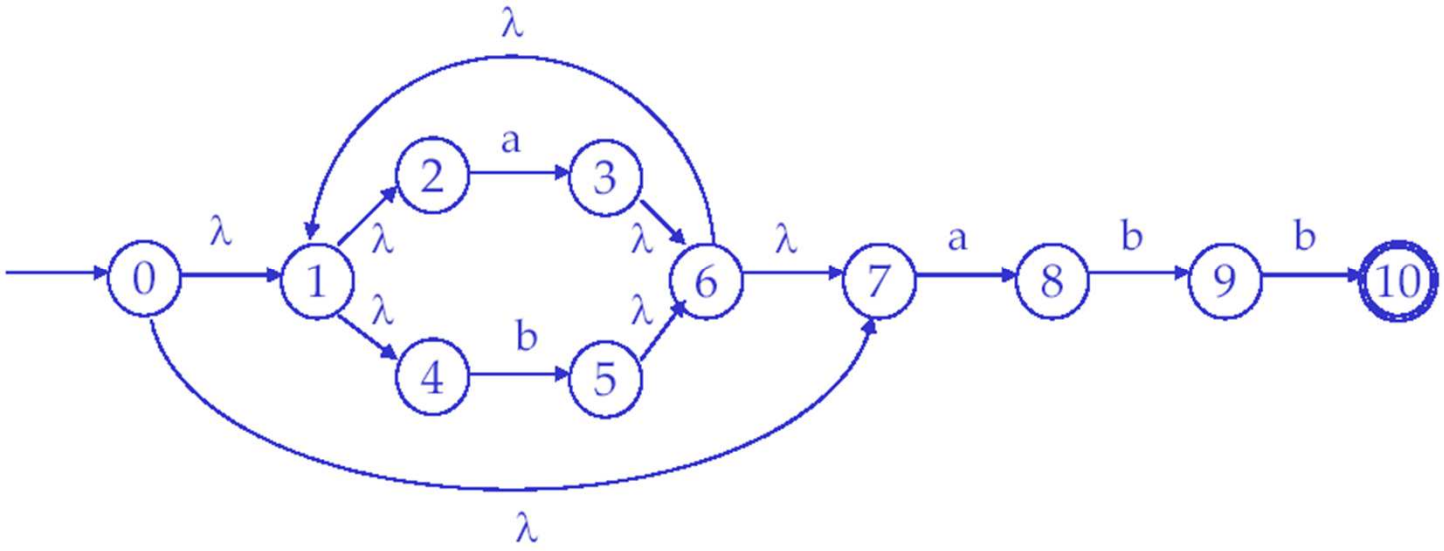
UNIVERSIDAD FRANCISCO DE VITORIA

```
al inicio, cerradura-e(s0) es el único estado dentro de estadosD
    y no está marcado;
while haya un estado no marcado T en estadosD do begin
    marcar T;
    for cada símbolo de estrada a do begin
            U := cerradura-e(mueve(T,a));
            if U no está en estadosD then
                    añadir U como estado no marcado a estadosD
            tranD[T,a]:=U
    end
end
```

Nota: la obtención de un U vacío significa patrón no reconocible o no perteneciente a la ER.

2. Gramáticas regulares y reconocedores: autómatas finitos

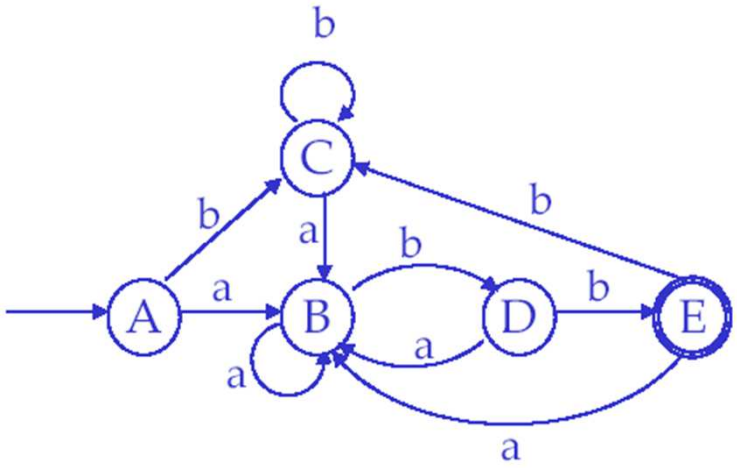
Ejemplo: transformación de AFN para (a|b)*abb en AFD.



- Estado inicial: A = { 0, 1, 2 , 4, 7 }
- Transición de A con 'a': B = { 1, 2, 3, 4, 6, 7, 8 }
- Transición de A con 'b': C = { 1, 2, 4, 5, 6, 7 }

2. Gramáticas regulares y reconocedores: autómatas finitos

- Transición de B con ‘a’: B
- Transición de B con ‘b’: D = { 1, 2, 4, 5, 6, 7, 9 }
- Transición de C con ‘a’: B
- Transición de C con ‘b’: C
- Transición de D con ‘a’: B
- Transición de D con ‘b’: E = { 1, 2, 4, 5, 6, 7, 10 } (final)
- Transición de E con ‘a’: B
- Transición de E con ‘b’: C



UNIVERSIDAD FRANCISCO DE VITORIA

Minimización de estados de un AFD

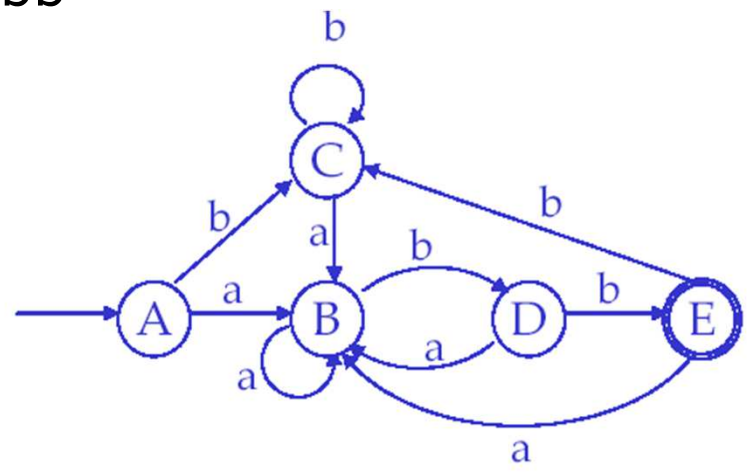
Algoritmo

1. Crear una partición con dos grupos: los estados finales y los no finales
2. Para cada grupo con varios estados, dividir el grupo en subgrupos tales que dos estados, *s y t*, estén en el mismo subgrupo si y solo si para cada símbolo a, si existe la transición desde *s* con el símbolo a hacia un estado de un cierto grupo, entonces debe existir la transición desde *t* con el símbolo a hacia un estado del mismo grupo.
3. Repetir el paso 2 hasta que no se dividan más grupos
4. Cada grupo representa un estado en el AFD minimizado.
5. Eliminar los estados no alcanzables desde el estado inicial y los que no tengan transiciones que puedan conducir a un estado final.

2. Gramáticas regulares y reconocedores: autómatas finitos

UNIVERSIDAD FRANCISCO DE VITORIA

Ejemplo: (a|b)*abb

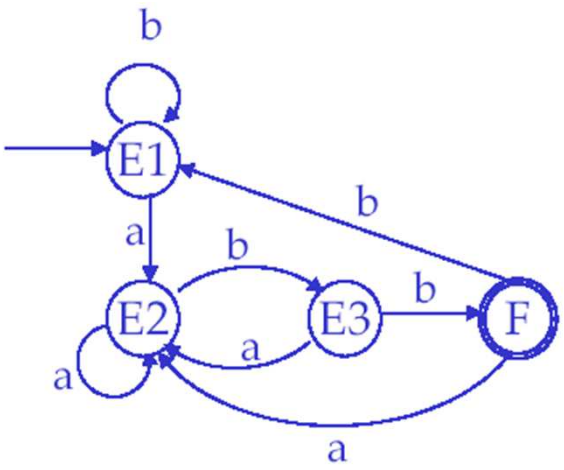


	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

		a	b
E1	A	E1	E1
	B	E1	E1
	C	E1	E1
	D	E1	F
F	E	E1	E1

		a	b
E1	A	E1	E1
	B	E1	E2
	C	E1	E1
E2	D	E1	F
F	E	E1	E1

2. Gramáticas regulares y reconocedores: autómatas finitos



	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

		a	b
E1	A	E2	E1
	C	E2	E1
E2	B	E2	E3
E3	D	E2	F
F	E	E2	E1

- No existen estados no alcanzables
- No existen estados que no conduzcan a un estado final

UNIVERSIDAD FRANCISCO DE VITORIA

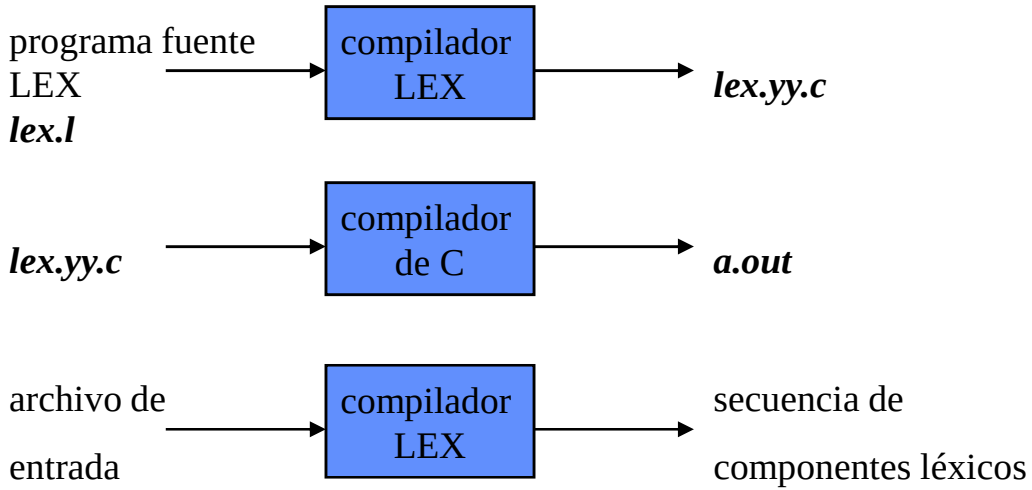
Comparación AFD vs AFN

- Tienen la misma capacidad descriptiva
- Número de estados (memoria ocupada):
  - en un AFN crece linealmente con el tamaño de la expresión. Una expresión de tamaño r puede ser reconocida por un AFN de 2r estados
  - en un AFD crece exponencialmente con el tamaño de la expresión. En el peor caso, una expresión de tamaño r puede necesitar un AFD con 2^r estados
- El tiempo de análisis:
  - en un AFD es de orden O(n)
  - en un AFN es de orden O(n*r)
  - n (tamaño de la cadena), r (tamaño de la expresión)

UNIVERSIDAD FRANCISCO DE VITORIA

Generación automática del analizador léxico

- LEX:
  - Creado por M. E. Lesk y E. Schmidt (Bell Laboratories).
  - Genera analizadores léxicos para C, Fortran, Raftor.
  - Hay versiones para Unix, Linux, DOS, etc.
- Funcionamiento de LEX



2. Gramáticas regulares y reconocedores: autómatas finitos

UNIVERSIDAD FRANCISCO DE VITORIA

- Estructura del fichero de entrada de LEX

declaraciones (opcional)
%%
reglas de traducción de las expresiones regulares
%%
funciones auxiliares (opcional)

- Estructura de reglas de traducción para expresiones regulares.

p1 {acción 1}
p2 {acción 2}
p3 {acción 3}

2. Gramáticas regulares y reconocedores: autómatas finitos

UNIVERSIDAD FRANCISCO DE VITORIA

- LEX: expresiones regulares
  - *|: disyunción*
  - *( ): agrupación de expresiones regulares*
  - **: repetición de un patrón cero o más veces.*
  - *+: repetición de un patrón una o más veces.*
  - *?: el patrón puede aparecer cero o una vez.*
  - *" ": delimitadores de cadenas*
  - *.: cualquier carácter distinto del salto de línea (\n).*
  - *\n: salto de línea*
  - *$: carácter de final de línea*
  - *[ ]: delimitadores de clases de caracteres*
  - *ˆ : inicio de línea y complementario de una clase.*
- Si se antepone la barra \ delante de un símbolo especial entonces sólo se representa a sí mismo:
  - \. → sólo representa el punto.

2. Gramáticas regulares y reconocedores: autómatas finitos

UNIVERSIDAD FRANCISCO DE VITORIA

- Ejemplo de código LEX

```
int num_lines = 0, num_chars = 0;

%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;

%%
main()
{
    yylex();
    printf( "# of lines = %d, # of chars = %d\n",num_lines,
    num_chars );
}
```

3. Gramáticas independientes de contexto y analizadores sintácticos

UNIVERSIDAD FRANCISCO DE VITORIA

Características del analizador sintáctico

- Lee componentes léxicos (tokens)
- Comprueba que el orden de estos corresponde a la sintaxis predeterminada
- Genera errores en caso de que el flujo de tokens no responda a la sintaxis
- Genera árboles de análisis sintáctico
- Se suele conocer como "Parser"
- El análisis sintáctico desarrolla el esqueleto de toda la fase de análisis
- Utiliza el analizador léxico como una rutina dentro del análisis sintáctico (getNextToken())
- Integra el análisis semántico como un conjunto de rutinas a ejecutar durante la comprobación de la sintaxis

3. Gramáticas independientes de contexto y analizadores sintácticos

UNIVERSIDAD FRANCISCO DE VITORIA

- Las gramáticas regulares no son adecuadas para describir la sintaxis de lenguajes de programación
  - [Ejemplo] sintaxis de llaves abiertas y cerradas: {{{{ a }}}}
    - El número de llaves abiertas y cerradas debe ser el mismo
    - Es indispensable para analizar la sintaxis de C, C++ y Java (p. e.)
  - [Expresión regular] : ('{')* a ('}')*
    - Puede generar un número de llaves diferente
- Las gramáticas G-2 sí permiten este tipo de sintaxis. Estás gramáticas requieren autómatas pila (recursividad de reglas).
  - <A> → '{' <A> '}'
  - <A> → a

3. Gramáticas independientes de contexto y analizadores sintácticos

UNIVERSIDAD FRANCISCO DE VITORIA

- El análisis sintáctico se especifica normalmente mediante una gramática libre de contexto.
  - Ejemplo: S -> α S β | ε es libre de contexto
  - Ejemplo: $\alpha\ \beta\ \alpha$ no es una cadena que se pueda derivar con una gramática libre de contexto (una declaración de variable antes de su uso).
  - $L_3=\{a^n b^n c^n \mid n > 0\}$ no es independiente
  - $L_4=\{wcw^R \mid w^R$ es w invertido$\}$ es independiente

3. Gramáticas independientes de contexto y analizadores sintácticos

UNIVERSIDAD FRANCISCO DE VITORIA

Gramáticas independientes de contexto

- Una gramática independiente de contexto consta de terminales, no terminales, un símbolo inicial y producciones.
  - **Terminales:** Son símbolos básicos que forman las cadenas. "Componente léxico" es sinónimo de "terminal".
  - **No terminales**: son variables sintácticas que denotan conjuntos de cadenas. Los no terminales definen conjuntos de cadenas que ayudan a definir el lenguaje generado por la gramática. También imponen una estructura jerárquica sobre el lenguaje que es útil tanto para el análisis sintáctico como para la traducción.

3. Gramáticas independientes de contexto y analizadores sintácticos

UNIVERSIDAD FRANCISCO DE VITORIA

- En una gramática, un no terminal es considerado como símbolo inicial, y el conjunto de cadenas que representa el lenguaje definido por la gramática

- Las producciones de una gramática especifican cómo se pueden combinar los terminales para formar cadenas. Cada producción consta de un no terminal, seguido por una flecha, seguida por una cadena de no terminales y terminales.

3. Gramáticas independientes de contexto y analizadores sintácticos

UNIVERSIDAD FRANCISCO DE VITORIA

Convenciones de notación

1. Símbolos terminales:
   1. primeras letras del alfabeto en minúsculas.
   2. símbolos y dígitos
   3. cadenas minúsculas, en negrita o subrayadas.

2. Símbolos no terminales:
   1. primeras letras del alfabeto en mayúsculas.
   2. La S suele representar el símbolo inicial.

3. Las letras griegas α, β, ... representan cadenas de símbolos gramaticales.

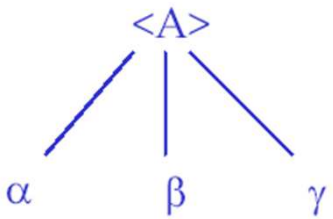
3. Gramáticas independientes de contexto y analizadores sintácticos

UNIVERSIDAD FRANCISCO DE VITORIA

Árboles de análisis sintáctico y derivaciones

- Los nodos corresponden a símbolos no terminales
- Las hojas corresponden a símbolos terminales y no terminales
- La ramificación describe una producción Ejemplo de producción: <A> $\rightarrow \alpha\ \beta\ \gamma$

Ejemplo de árbol de análisis sintáctico



3. Gramáticas independientes de contexto y analizadores sintácticos

UNIVERSIDAD FRANCISCO DE VITORIA

- Derivación de árboles de análisis sintáctico
  - Sucesión de árboles de análisis sintáctico
  - El primer árbol representa el símbolo inicial
  - Cada árbol se diferencia del anterior en la expansión de una hoja no terminal por medio de una producción
  - Las hojas del árbol final son todas símbolos terminales
  - Las formas sentenciales se obtienen leyendo las hojas de izquierda a derecha

Ejemplo de derivación de árboles de análisis sintáctico

- Gramática:

  <E> → <E> + <E>

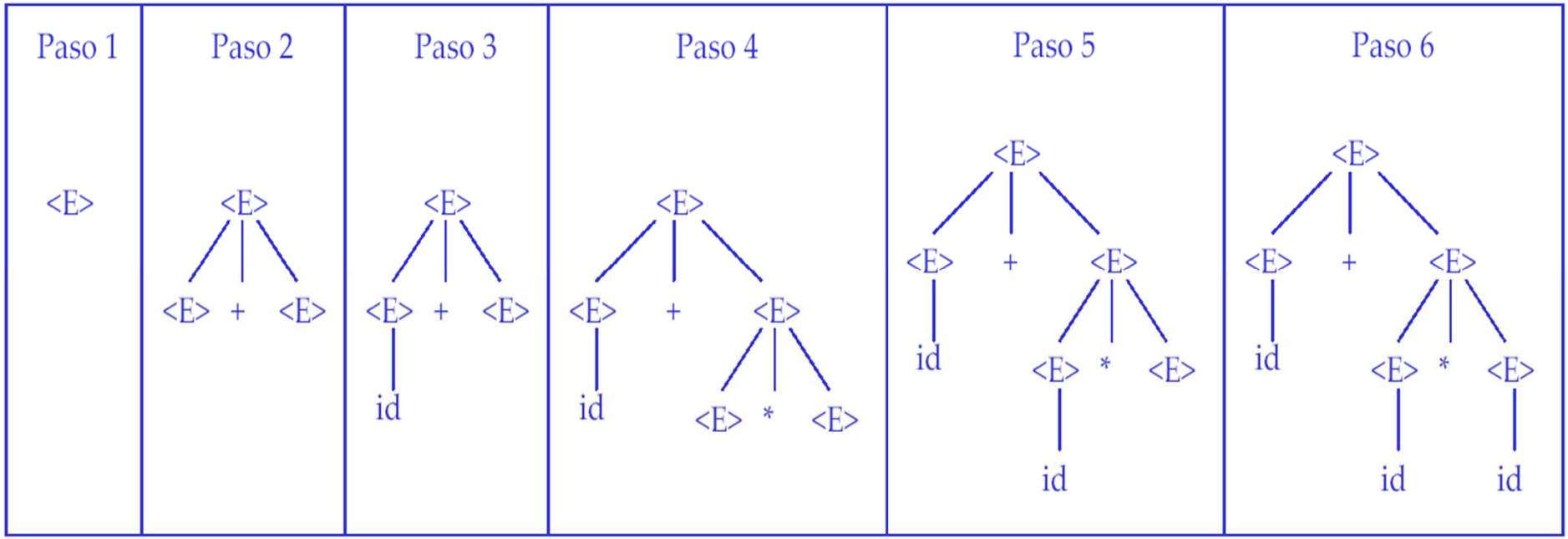
  <E> → <E> * <E>

  <E> → (<E>)

  <E> → id
- Sentencia:

  id + id * id

3. Gramáticas independientes de contexto y analizadores sintácticos



3. Gramáticas independientes de contexto y analizadores sintácticos

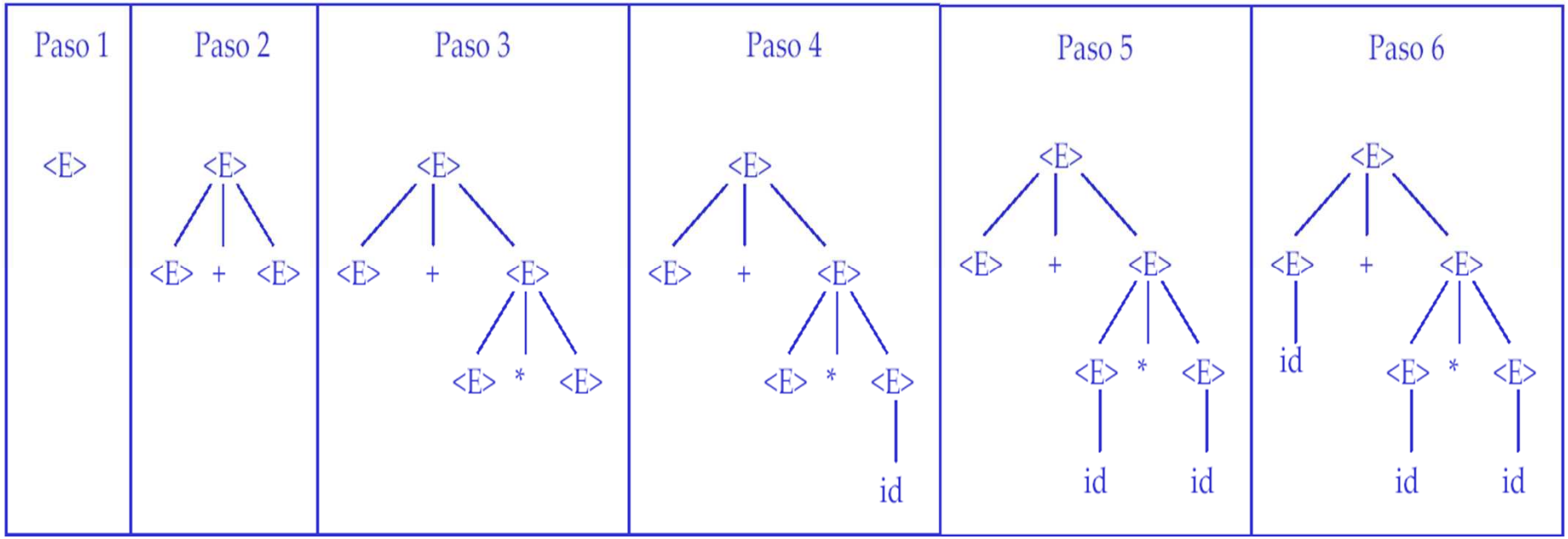
UNIVERSIDAD FRANCISCO DE VITORIA

- Derivación canónica:
  - Aquella que sigue un orden predeterminado para sustituir las hojas no terminales
- Derivación canónica a la izquierda:
  - Se sustituye siempre la hoja no terminal que se encuentre más a la izquierda
  - El ejemplo anterior es una derivación a la izquierda
- Derivación canónica a la derecha:
  - Se sustituye siempre la hoja no terminal que se encuentre más a la derecha

3. Gramáticas independientes de contexto y analizadores sintácticos

UNIVERSIDAD FRANCISCO DE VITORIA

Ejemplo de derivación canónica a la derecha



3. Gramáticas independientes de contexto y analizadores sintácticos

UNIVERSIDAD FRANCISCO DE VITORIA

Ambigüedad

- Una gramática es ambigua si existe alguna cadena a la que se pueda asociar dos árboles de análisis sintáctico diferentes
- Las gramáticas ambiguas no son adecuadas para especificar lenguajes de programación, ya que árboles sintácticos diferentes generan un código diferente.
- Demostrar que una gramática no es ambigua es muy difícil
- Existen construcciones gramaticales que producen ambigüedad.
- Si se usan estas construcciones es fácil demostrar que una gramática es ambigua.

3. Gramáticas independientes de contexto y analizadores sintácticos

UNIVERSIDAD FRANCISCO DE VITORIA

- Construcciones que provocan ambigüedad:
  - Reglas de la forma
    - <E> → <E> .... <E>
  - Gramáticas con ciclos
    - <S> → <A>
    - <S> → a
    - <A> → <S>
  - Conjuntos de reglas con caminos alternativos entre dos puntos
    - <S> → <A>
    - <S> → <B>
    - <A> → <B>

3. Gramáticas independientes de contexto y analizadores sintácticos

UNIVERSIDAD FRANCISCO DE VITORIA

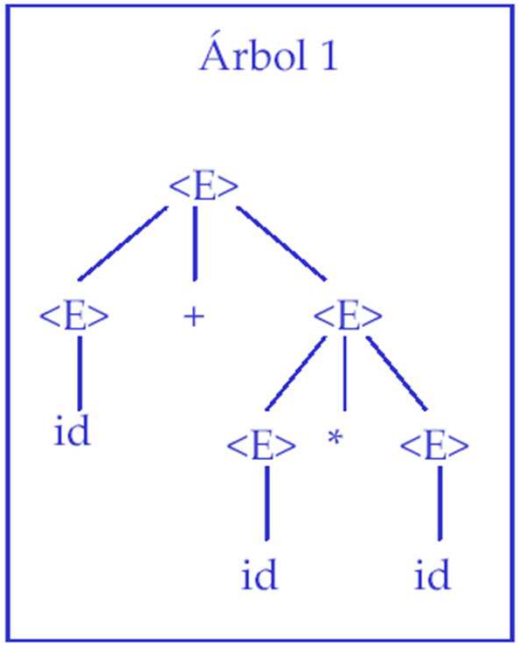
- Producciones recursivas en las que la regla no recursiva pueda producir la cadena vacía
  - <S> $\rightarrow$ <H><R><S>
  - <S> $\rightarrow$ s
  - <H> $\rightarrow$ h | ε
  - <R> $\rightarrow$ r | ε

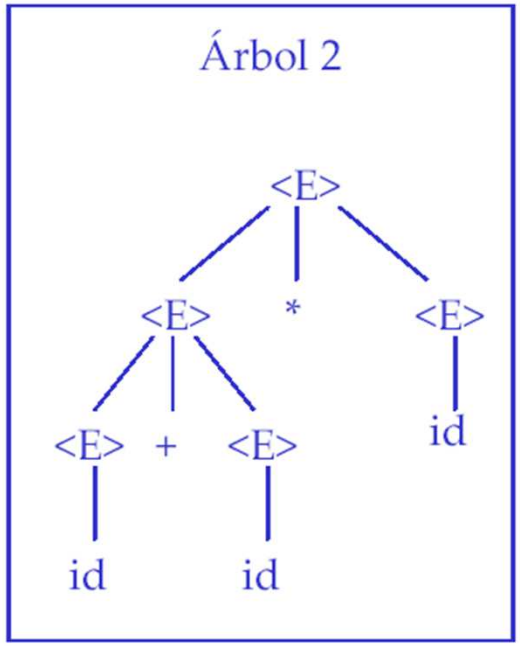
- Símbolos no terminales que puedan producir la misma cadena y la cadena vacía, si aparecen juntos
  - <S> $\rightarrow$ <H><R>
  - <H> $\rightarrow$ h | ε
  - <R> $\rightarrow$ h | r | ε

3. Gramáticas independientes de contexto y analizadores sintácticos

UNIVERSIDAD FRANCISCO DE VITORIA

Ejemplos de ambigüedad: *id* + *id* * *id*





UNIVERSIDAD FRANCISCO DE VITORIA

Supresión de la ambigüedad

- A veces, una gramática ambigua se puede reescribir para eliminar la ambigüedad. Como ejemplo, eliminaremos la ambigüedad en la gramática con ***else ambiguo***.

prop → <u>if</u> expr <u>then</u> prop
| <u>if</u> expr <u>then</u> prop <u>else</u> prop
| *otra*

<u>if</u> E1 <u>then</u> S1 <u>if</u> E2 <u>then</u> S2 <u>else</u> S3

3. Gramáticas independientes de contexto y analizadores sintácticos

UNIVERSIDAD FRANCISCO DE VITORIA

- La anterior frase tiene 2 árboles de derivación distintos donde el else puede quedar asociado al primer o segundo if.

- En general, en casi todos los lenguajes se asocia el else a su if más cercano. Podríamos reescribir la gramática, teniendo en cuenta reglas de producción de proposición para ifs emparejados y no emparejados

prop → prop_emparejada
|prop_no_emparejada
prop_emparejada → if expr then prop_emparejada else prop_emparejada
| *otra*
prop_no_emparejada → if expr then prop
|if expr then prop_emparejada else prop_no_emparejada

3. Gramáticas independientes de contexto y analizadores sintácticos

UNIVERSIDAD FRANCISCO DE VITORIA

Recursividad

- Una gramática es recursiva si a partir de un símbolo no terminal se puede obtener una forma sentencial que incluya el mismo símbolo no terminal
  - <A> → a <A> b
- Una gramática es recursiva a la izquierda si en alguna recursividad, el símbolo terminal aparece a la izquierda de la forma sentencial
  - <A> → <A> b
- Una gramática es recursiva a la derecha si en alguna recursividad, el símbolo terminal aparece a la derecha de la forma sentencial
  - <A> → a <A>

Eliminación de la recursión por la izquierda

- Los métodos de análisis sintáctico descendente no pueden manejar gramáticas recursivas por la izquierda, así que se necesita una transformación que la elimine.
  - Dada la siguiente regla de producción recursiva:
    <A> $\rightarrow$ <A>α | β
  - Puede ser transformada al conjunto de reglas equivalente no recursiva:
    <A> $\rightarrow$ β <A'>
    <A'> $\rightarrow$ α <A'> | ε

UNIVERSIDAD FRANCISCO DE VITORIA

Eliminación de la recursión por la izquierda (II)

- Algoritmo (eliminación de recursividad inmediata por la izquierda):

Dada una gramática G de la forma:

$$<A> \rightarrow A\ \alpha_1 \mid A\ \alpha_2 \mid \ldots A\ \alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \beta_n$$

transformarla a de la siguiente manera:

$$<A> \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \beta_n A'$$

$$<A'> \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \alpha_m A' \mid \varepsilon$$

- Problema: Este método no elimina la recursividad de 2 ó más pasos:

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid \varepsilon$$

UNIVERSIDAD FRANCISCO DE VITORIA

Eliminación de la recursión por la izquierda (III)

- Algoritmo (eliminación de la recursividad general):

Entrada: La gramática G sin ciclos ni producciones ε

Salida: Una gramática equivalente sin recursión por la izquierda

Ordénese los no terminales en un orden $A_1, A_2, \ldots, A_n$

for i:=1 **to** n **do**

begin

for j:=1 **to** i-1 **do**

sustituir cada producción de la forma $<A_i> \rightarrow <A_j>\alpha$ por las producciones $<A_i> \rightarrow \beta_1\alpha \mid \beta_2\alpha \mid \ldots \beta_k\alpha$ tal que $<A_j> \rightarrow \beta_1 \mid \beta_1 \mid \ldots \beta_k$ son todas las producciones de A_j.

Eliminar la recursividad inmediata por la izquierda entre las producciones de A_i

end

UNIVERSIDAD FRANCISCO DE VITORIA

Eliminación de la recursión por la izquierda (IV)

Ejercicios:

- *Eliminar recursividad de*

  E->E+T|T

  *T->T*F|F*

  F->(E)|num|id

- *Eliminar la recursividad de*

  S -> (L) | a

  L -> L,S|S

- *Eliminar la recursividad de*

  *EXPRESION::= EXPRESION * TERMINO | EXPRESION + TERMINO| TERMINO*

UNIVERSIDAD FRANCISCO DE VITORIA

Factorización por la izquierda (I)

- Es una transformación gramatical útil para producir una gramática adecuada para el análisis sintáctico predictivo.
- La idea básica que cuando no está claro cuál de dos producciones alternativas se deben utilizar, se pueden reescribir las producciones para retrasar la decisión hasta que haya suficientes terminales que permitan decidir sin alternativas.

prop → <u>if</u> expr <u>then</u> prop
| <u>if</u> expr <u>then</u> prop <u>else</u> prop

UNIVERSIDAD FRANCISCO DE VITORIA

Factorización por la izquierda (II): Algoritmo

Entrada: La gramática G

Salida: Una gramática equivalente factorizada por la izquierda

Método: Para cada no terminal A, encuéntrese el prefijo α más largo común a dos o más alternativas. Si $\alpha != \varepsilon$, es decir, existe un prefijo común no trivial, sustitúyanse todas las producciones de A, $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \ldots | \alpha\beta_n | \gamma$ donde γ representa todas las alternativas que no comienzan por α, por

$A \rightarrow \alpha A' | \gamma$

$A' \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n$

Aplicar repetidamente esta transformación hasta que no haya dos alternativas para un no terminal con un prefijo común.

4. Gramáticas LL(k) y analizadores descendentes

UNIVERSIDAD FRANCISCO DE VITORIA

Factorización por la izquierda (III): Ejemplo

P → iEtP| iEtPeP|a
E → b

factorizado como:

P → iEtPP'| a
P' → eP|ε
E → b

- **Nota: No confundir factorización con supresión de ambigüedad**

UNIVERSIDAD FRANCISCO DE VITORIA

Factorización por la izquierda (IV): Ejemplo 2

$E \rightarrow T+E \mid T$
$T \rightarrow int \mid int * T \mid (\ E)$

factorizado como:

$E \rightarrow TX$
$X \rightarrow +E \mid \xi$
$T \rightarrow (\ E) \mid int\ Y$
$Y \rightarrow * T \mid \xi$

3. Gramáticas independientes de contexto y analizadores sintácticos

UNIVERSIDAD FRANCISCO DE VITORIA

Tipos de análisis sintácticos

- Algoritmos de resolución general
  - Algoritmo de Cocke-Younger-Kasami: Orden $|x|^3$
  - Algoritmo de Early: $O(|x|^3)$ en gramáticas ambiguas y $O(|x|^2)$ en gramáticas no ambiguas
  - Para una compilación en un tiempo razonable es necesario un orden lineal $O(|x|)$

- Algoritmos de orden lineal
  - Imponen restricciones a las gramáticas a analizar
  - **Análisis descendente**
  - **Análisis ascendente**

4. Gramáticas LL(k) y analizadores descendentes

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis descendente

- Se construye el árbol de análisis sintáctico partiendo de la raíz hacia las hojas. Se realiza la derivación canónica por la izquierda.
- Se estudia los siguientes tokens a analizar para decidir la regla a expandir.
- Lookahead: Nº de tokens necesario para realizar la elección de la regla a expandir.

4. Gramáticas LL(k) y analizadores descendentes

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis descendente (II)

- Existen dos formas de implementarlos:
  - **Análisis descendente recursivo (ASD recursivo)**: Es la manera más sencilla, implementándose con una función recursiva aprovechando la recursividad de la gramática.
  - **Análisis descendente predictivo (ASD predictivo)**: Para aumentar la eficiencia, evitando los retrocesos, se predice en cada momento cuál de las reglas sintácticas hay que aplicar para continuar el análisis.

- En la práctica no se emplea nunca el recursivo (o con retroceso) debido a diversos inconvenientes.

4. Gramáticas LL(k) y analizadores descendentes

UNIVERSIDAD FRANCISCO DE VITORIA

ASD Recursivo

- Mediante un método de **backtracking** se van probando todas las opciones de expansión para cada no-terminal de la gramática hasta encontrar la correcta.

- Cada **retroceso en el árbol sintáctico tiene asociado un retroceso en la entrada**: se deben eliminar todos los terminales y no terminales correspondientes a la producción que se "elimina" del árbol.

- Si el terminal obtenido como consecuencia de probar con una opción de las varias de una producción no coincide con el componente léxico leído en la entrada, hay que retroceder.

4. Gramáticas LL(k) y analizadores descendentes

UNIVERSIDAD FRANCISCO DE VITORIA

ASD Recursivo (II). Algoritmo

1. Se colocan las reglas en orden, de forma que si la parte derecha de una producción es prefijo de otra, esta última se sitúa detrás.
2. Se crea el nodo inicial con el axioma y se considera nodo activo.
3. Para cada nodo activo A:
   - Si A es un no-terminal, se aplica la primera producción asociada a A.
     - El nodo activo pasa a ser el hijo izquierdo.
     - Cuando se terminan de tratar todos los descendientes, el siguiente nodo activo es el siguiente hijo por la izquierda.
   - Si A es un terminal:
     - Si coincide el símbolo de la entrada, se avanza el puntero de entrada y el nodo activo pasa a ser el siguiente "hermano" de A.
     - Si no, se retrocede en el árbol hasta el anterior no-terminal (y en la entrada si se ha reconocido algún componente léxico mediante la producción que se elimina) y se prueba la siguiente producción.
     - Si no hay más producciones para probar, se retrocede hasta el anterior no-terminal y se prueba con la siguiente opción de éste.
4. Si se acaban todas las opciones del nodo inicial, error sintáctico. Si por el contrario se encuentra un árbol para la cadena de entrada, éxito.

4. Gramáticas LL(k) y analizadores descendentes

UNIVERSIDAD FRANCISCO DE VITORIA

ASD Recursivo (III). Ejemplo

- Gramática:

① $E \rightarrow T + E$
② $E \rightarrow T$
③ $T \rightarrow F * T$
④ $T \rightarrow F$
⑤ $F \rightarrow a$
⑥ $F \rightarrow b$
⑦ $F \rightarrow (E)$

- Cadena de entrada: (a + b) * a + b
- Parser a izquierdas: 1-3-7-1-4-5-2-4-6-4-5-2-4-6

UNIVERSIDAD FRANCISCO DE VITORIA

ASD Recursivo (IV). Ejemplo

- Método de trabajo:

(a + b) * a + b

① E → T + E
② E → T
③ T → F * T
④ T → F
⑤ F → a
⑥ F → b
⑦ F → (E)

forma_sentencial	Pila (de reglas utilizadas)
E	1
T + E	1-3
F * T +E	1-3-5
a * T + E	1-3
F * T +E	1-3-6
b * T + E	1-3
F * T +E	1-3-7
(E) * T +E	1-3-7-1
(T + E) * T + E	1-3-7-1-3
(F * T + E) * T + E	1-3-7-1-3-5
(a * T + E) * T + E	1-3-7-1-3
(F * T + E) * T + E	1-3-7-1-3-6
(b * T + E) * T + E	1-3-7-1-3
(F * T + E) * T + E	1-3-7-1-3-7
((E) * T + E) * T + E	1-3-7-1-3
(F * T + E) * T + E	1-3-7-1
(T + E) * T + E	1-3-7-1-4
(F + E) * T + E	1-3-7-1-4-5

4. Gramáticas LL(k) y analizadores descendentes

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos

- En muchos casos, escribiendo la gramática con cuidado, eliminando su recursión por la izquierda y factorizando por la izquierda, la gramática resultante puede ser analizable con un analizador descendente predictivo.

- Un analizador predictivo no requerirá retrocesos ni cambios sobre las reglas de derivación aplicadas.

prop → <u>if</u> expr <u>then</u> prop <u>else</u> prop
| <u>while</u> expr <u>do</u> prop
| <u>begin</u> lista_props <u>end</u>

Las palabras <u>if</u>, <u>while</u> y <u>begin</u> indican la alternativa única de derivación cuando se trata una proposición.

4. Gramáticas LL(k) y analizadores descendentes

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (II)

- El análisis predictivo acepta gramáticas LL(k)
  - L significa explorar la entrada de izda-a-dcha.
  - L significa "derivación mas a la izquierda"
  - K significa "predicción basada en siguientes k tokens".

- En la práctica se emplea LL(1).

4. Gramáticas LL(k) y analizadores descendentes

UNIVERSIDAD FRANCISCO DE VITORIA

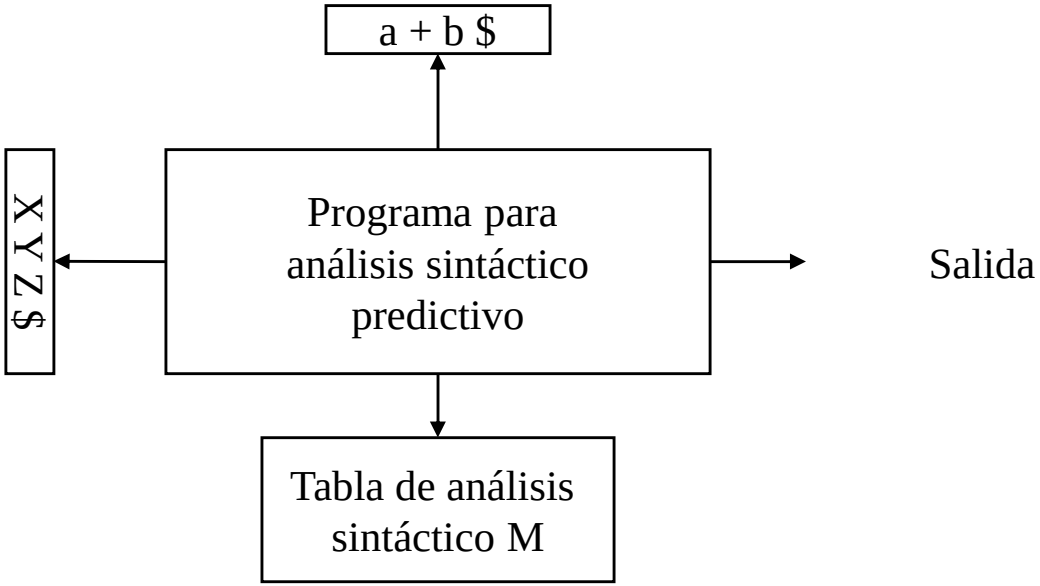
Análisis sintácticos predictivos (III)

- Los lenguajes LL(1) cumplen además las siguientes propiedades:
  - Para cada no terminal y tokens hay sólo una producción que puede conducir al éxito.
  - Se puede especificar en una tabla 2D
    - Una dimensión para no terminales
    - Una dimensión para el próximo token
    - Entradas a la tabla que contienen una producción

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (IV)

- A partir del modelo anterior, es posible generar un analizador predictivo no recursivo, mediante el uso de una estructura PILA y una tabla de análisis sintáctico.



UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (V): Tabla de símbolos

- Se trata de una matriz M[N,T] donde se representan las producciones a expandir en función del estado actual del análisis y del símbolo de la entrada.
  - Nota: Los cuadros en blanco darían lugar a errores, p.e., con S y en la entrada b, supone un error.

	a	b	c	d	e	$
S	S→BA			S→BA		
A		A→ bSC			A→ε	A→ ε
B	B→DC			B→ DC		
C		C→ ε	C→cDC		C→ε	C→ ε
D	D→ a			D → dSe		

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (VI): Algoritmo

- *Entrada*: Una cadena *w* y una tabla de análisis sintáctico *M* para la gramática *G*.
- *Salida:* Si *w* está en L(*G*), una derivación por la izquierda de *w*; en caso contrario, una indicación de error.
- *Método:* Al principio, el analizador sintáctico está en una configuración en la que tiene a $S en la pila con , el símbolo inicial en el tope, y *w$* en el *buffer* de entrada. Se sigue el siguiente procedimiento:

4. Gramáticas LL(k) y analizadores descendentes

UNIVERSIDAD FRANCISCO DE VITORIA

apuntar *ptr* al primer símbolo de *w$*

repeat

Sea ***X*** el símbolo inicial de la cima de la pila y ***a*** el símbolo apuntado por *ptr*;

if *X* es un terminal o *$* **then**

if *X* = *a* ***then***

extraer X de la pila y avanzar *ptr*.

else *error();*

else

if M[*X*,*a*] = $X \rightarrow Y_1\ Y_2 \ldots Y_k$ **then begin**

extraer *X* de la pila;

meter $Y_k,\ Y_{k-1} \ldots Y_1$ en la pila, con Y_1 en la cima;

emitir la producción $X \rightarrow Y_1\ Y_2 \ldots Y_k$

end

else *error();*

until *X* = *$* /* la pila está vacía */

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (VIII) Ejemplo

E--> T E'
E' --> + T E' | e
T --> F T'
T' --> * F T' | e
F --> (E) | id

No terminal	Simbolo de entrada					
	id	+	*	(	)	$
E	E--> TE'			E-->TE'		
E'		E'-->+TE'			E'-->e	E'-->e
T	T-->FT'			T-->FT'		
T'		T'-->e	T'-->*FT'		T'-->e	T'-->e
F	F-->id			F-->(E)		

Ejercicio: realizar la simulación del reconocimiento no recursivo para la cadena de entrada "(id+id)*id$".

Pila	Entrada	Salida
$E	id+id*id$	
$E'T	id+id*id$	E-->TE'
$E'T'F	id+id*id$	T-->FT'
$E'T'id	id+id*id$	F-->id
$E'T'	+id*id$	
$E'	+id*id$	T'-->e
$E'T+	+id*id$	E'-->+TE'
$E'T	id*id$	
$E'T'F	id*id$	T-->FT'
$E'T'id	id*id$	F-->id
$E'T'	*id$	
$E'T'F*	*id$	T'-->*FT'
$E'T'F	id$	
$E'T'id	id$	F-->id
$E'T'	$	
$E'	$	T'-->e
$	$	T'-->e

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (IX): Comprobar si una gramática es LL(1)

- Para que una gramática sea LL(1) ha de cumplir:
  1. Para las producciones de la forma $A \rightarrow \alpha_1 | \alpha_2 \ldots | \alpha_n$, los conjuntos Primero(α_1), Primero(α_2),..., Primero(α_n) son disjuntos.
  2. A lo sumo un α_i puede derivar la cadena vacía ξ.
  3. Si el no terminal A es tal que {ξ} pertenece a PRIMERO(A), entonces los conjuntos PRIMERO(A) y SIGUIENTE(A) son disjuntos.

4. Gramáticas LL(k) y analizadores descendentes

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (X): Comprobar si una gramática es LL(1)

- Dado un símbolo de la gramática A, su conjunto **Primeros** es el conjunto de terminales por los que comienza cualquier frase que se genere a partir de A mediante derivaciones por la izquierda de las producciones.
  - $a \in$ Primero(α_p) si $a \in (T \cup \xi) \mid \alpha_p \rightarrow a\ \beta$
- Algoritmo de cálculo del conjunto primero:
  - Si $A \in T$ => Prim(A) = A
  - Si $A \rightarrow \xi$ => $\xi \in$ Prim(A)
  - Si $A \rightarrow Y_1 Y_2 \ldots Y_n$ => $\{Prim(Y_1) - \xi\} \in$ Prim(A)
  - Si $\xi \in$ Prim (Y_1) => $\{Prim\ (Y_2) - \xi\} \in$ Prim(A)
  - Si $\xi \in$ Prim (Y_2) => $\{Prim\ (Y_3) - \xi\} \in$ Prim(A)
  - …
  - Si i, $\xi \in$ Prim(Y_i) => $\xi \in$ Prim(A)

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (XI): Comprobar si una gramática es LL(1)

- Dado un símbolo **A**, su conjunto **Siguientes** es el conjunto de terminales que aparecen inmediatamente después de A en alguna sentencia por la que se pasa al realizar una derivación por la izquierda.
  - a $\in$ Siguientes(A) si a $\in$ (Vt U {\$}) | $\exists$ S=> …Aa…
- **Algoritmo:**
  - Siguiente (S) = \$, siendo S el axioma.
  - Para cada símbolo no-terminal **A** de la gramática, añadir elementos aplicando las siguientes reglas:
    - Si $\exists$ X $\rightarrow$ αAB => {Primero(B) – ξ} $\in$ Siguiente(A)
    - Si ($\exists$ B $\rightarrow$ αA) $\vee$ (o) (B $\rightarrow$ αAC $\wedge$ (tal que) ξ $\in$ Primero(C)) => Siguiente(B)$\in$ Siguiente (A)
  - ξ nunca es elemento de un conjunto siguiente.

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (XII): Comprobar si una gramática es LL(1)

Gramática	*Primeros* de la regla	*Primeros* del no terminal
expr → expr + expr1 expr → expr - expr1 expr → expr1	{ (, *num*} { (, *num*} { (, *num*}	De entrada, *Primeros*(*expr*) = {∅} *Primeros* (*expr*) = *Primeros*(*expr*) ∪ *Primeros*(*expr1*) = { (, *num*}
expr1 → expr2 *expr1 expr1 → expr2 /expr1 expr1 → expr2	{ (, *num*} { (, *num*} { (, *num*}	*Primeros* (*expr1*) = *Primeros*(*expr2*) = { (, *num*}
expr2 → expr3 ^expr2 expr2 → expr3	{ (, *num*} { (, *num*}	*Primeros* (*expr2*) = *Primeros*(*expr3*) = { (, *num*}
expr3 → (expr) expr3 → num	{ (} { *num*}	*Primeros* (*expr3*) = { (, *num*}

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (XIII): Comprobar si una gramática es LL(1)

Gramática	Conjunto *Primeros*	Conjunto *Siguientes*
expr → expr + expr1 expr → expr - expr1 expr → expr1	*Primeros*(*expr*) = { (, *num*}	*Siguientes* (*expr*) = { +, -,), $ }
expr1 → expr2 * expr1 expr1 → expr2 / expr1 expr1 → expr2	*Primeros*(*expr1*) = { (, *num*}	*Siguientes* (*expr1*) = *Siguientes*(*expr*) = { +, -,), $ }
expr2 → expr3 ^ expr2 expr2 → expr3	*Primeros*(*expr2*) = { (, *num*}	*Siguientes* (*expr2*) = *Siguientes*(*expr1*) U { *, / } = { +, -,), $, *, / }
expr3 → (expr) expr3 → num	*Primeros*(*expr3*) = { (, *num*}	*Siguientes* (*expr3*) = *Siguientes*(*expr2*) U{ ^ } = { +, -,), $, *, /, ^ }

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (XIV): Comprobar si una gramática es LL(1)

- Gramática:

  $I \rightarrow TB$

  $T \rightarrow FC$

  $B \rightarrow +TB \mid -TB \mid \xi$

  $F \rightarrow x$

  $C \rightarrow *FC \mid /FC \mid \xi$

- Cálculo del conjunto de primeros

  $P(I) = \{x\}$

  $P(T) = \{x\}$

  $P(B) = \{\xi, +, -\}$

  $P(F) = \{x\}$

  $P(C) = \{\xi, *, /\}$

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (XV): Comprobar si una gramática es LL(1)

- Gramática:

  $I \rightarrow TB$

  $T \rightarrow FC$

  $B \rightarrow +TB \mid -TB \mid \xi$

  $F \rightarrow x$

  $C \rightarrow *FC \mid /FC \mid \xi$

- Cálculo del conjunto de siguientes

  $S(I) = \{\$\}$

  $S(T) = \{\$, +, -\}$

  $S(B) = \{\$\}$

  $S(F) = \{\$, *, /, +, -\}$

  $S(C) = \{\$, +, -\}$

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (XVI): Comprobar si una gramática es LL(1)

- Ejemplo:

$S \rightarrow ABe$	Primero(S) = {a,b,c,d}
$S \rightarrow C$	
$A \rightarrow dB$	Primero(A) = {a,c,d}
$A \rightarrow aS$	
$A \rightarrow c$	
$B \rightarrow AS$	Primero(B) = {a,b,c,d}
$B \rightarrow b$	
$C \rightarrow b$	Primero(C) = {b}

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (XVII): Comprobar si una gramática es LL(1)

- Ejercicio: calcular PRIMERO Y SIGUIENTE para la gramática:

  E->TE'

  E'->+TE'|-TE'| ξ

  T->FT'

  T'->*FT'|/FT'| ξ

  F->(E)| num | id

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (XVIII): Construir la tabla de análisis sintáctico

- Construir una tabla de análisis sintáctico T para la GLC G
- Para cada producción $A \rightarrow \alpha$ en G hacer:
  - Para cada terminal $b \in$ Primero(α) hacer:
    - $T[A,b] = \alpha$
  - Si $\alpha \rightarrow \xi$, para cada $b \in$ Siguiente(A) hacer
    - $T[A,b] = \alpha$

4. Gramáticas LL(k) y analizadores descendentes

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (XIX): Construir la tabla de análisis sintáctico

- Ejemplo: Comprobar si la siguiente gramática es LL(1) y construir la tabla, calculando todos los conjuntos Siguiente y Primero.

  $S \rightarrow cA$

  $A \rightarrow aB$

  $B \rightarrow b \mid \xi$

- Reconocer la cadena "cab" con el analizador construido.

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (XX): Construir la tabla de análisis sintáctico

- Pasos para ver si es LL(1):
  1. No es recursiva a izquierdas.
  2. $B \rightarrow b \mid \xi$
     $Prim(b) \cap Prim(\xi) = vacío$
     Si $\xi \in Prim(\xi) \Rightarrow Prim(b) \cap Sig(B) = \emptyset$
- Conjuntos primero:

  $Prim(S)=\{c\}$ $Prim(A)=\{a\}$ $Prim(B)=\{b, \xi\}$
- Conjuntos siguiente:

  $Sig(S)=\{\$\}$

  $Sig(A)=Sig(S)=\{\$\}$

  $Sig(B)=Sig(A)=\{\$\}$

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (XXI): Construir la tabla de análisis sintáctico

Para cada producción **A -> α:**

1. Para cada terminal **b** ∈ **Primero(α)** hacer: M [A,b] = A → α
2. Si **ξ**∈ **Primero(α)** , para cada terminal b∈ Siguiente(A), M [A,b] = A → α

- Tabla de análisis:

  S → cA

  A → aB

  B → b | ξ

  Prim(S)={c}, Prim(A)={a},Prim(B)={b, ξ}

  Sig(S)={$}, Sig(A)=Sig(S)={$}, Sig(B)=Sig(A)={$}

	c	a	b	$
S	S -> cA			
A		A -> aB		
B			B -> b	B -> ξ

UNIVERSIDAD FRANCISCO DE VITORIA

Análisis sintácticos predictivos (XXII): Construir la tabla de análisis sintáctico

Reconocer la cadena "cab"

	c	a	b	$
S	cA			
A		aB		
B			b	ξ

$S		cab$
$Ac	Reconocimiento	cab$
$A	A→aB	ab$
$Ba	Reconocimiento	ab$
$B	B→b	b$
$b	Reconocimiento	b$
$	éxito	$

4. Gramáticas LL(k) y analizadores descendentes

UNIVERSIDAD FRANCISCO DE VITORIA

Diagramas de Conway (diagramas de transición)

- Es posible describir el proceso de análisis sintáctico mediante diagramas de transición de estados. A diferencia de la etapa de análisis léxico, aquí existirá un diagrama para cada no terminal.
- La transición vendrá dada por la llegada de un nuevo componente léxico o símbolo no terminal.

- *Método:* Una vez eliminada la recursión por la izquierda y factorizado por la izquierda, para cada elemento no terminal A, realizar lo siguiente:
  - Créese un estado inicial y un estado final (de retorno)
  - Para cada producción $A \rightarrow X_1 | X_2 | \ldots | X_n$, créese un camino desde el estado inicial al estado final, con aristas etiquetadas $X_1, X_2, \ldots, X_n$.

4. Gramáticas LL(k) y analizadores descendentes

UNIVERSIDAD FRANCISCO DE VITORIA

Manejo de errores sintácticos

- Los programas pueden temer errores de muy diverso tipo:
  - léxicos, como escribir mal un identificador
  - sintácticos, como una expresión aritmética con paréntesis no equilibrados.
  - semánticos, como un operador aplicado a un operando incompatible.
  - lógicos, como una llamada infinitamente recursiva
- A menudo, gran parte de la detección y recuperación de errores de un compilador se centra en la fase de análisis sintáctico.

4. Gramáticas LL(k) y analizadores descendentes

UNIVERSIDAD FRANCISCO DE VITORIA

- Se debe informar de la presencia de errores con claridad y exactitud.
- Se debe recuperar de cada error con la suficiente rapidez como para detectar otros errores posteriores.
- No debe retrasar de manera significativa el procesamiento de programas correctos.
- Estrategias de recuperación de errores:
  - en modo de pánico
  - a nivel de frase
  - de producciones de error
  - de corrección global