

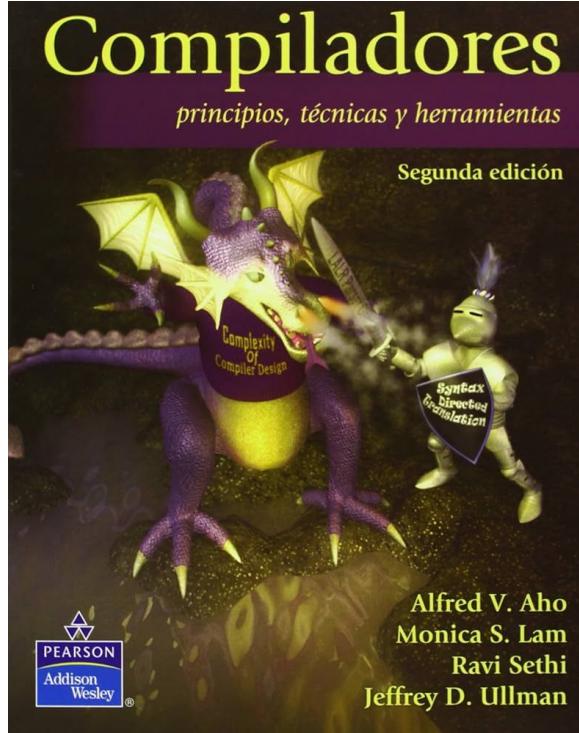
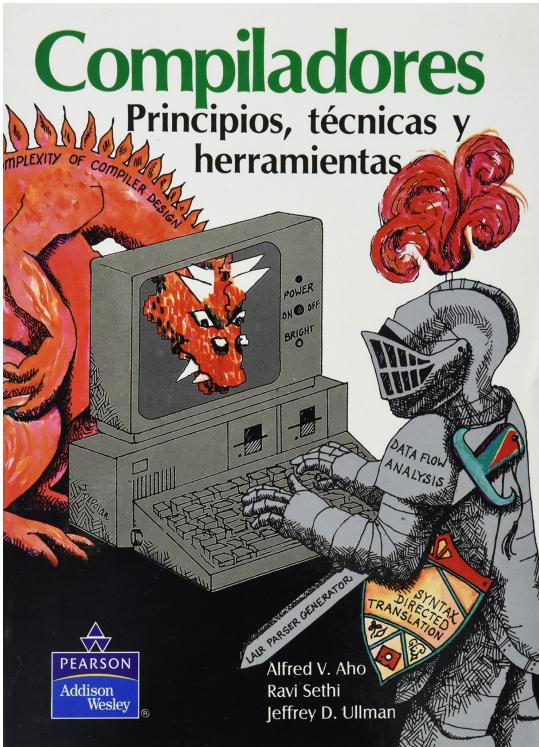


INTRODUCCIÓN A COMPILEDORES

El reto es vencer al dragón

Ing. Max Cerna

“Un buen idioma es aquel que la gente usa”



Agenda

1. Introducción
2. Acerca de Compiladores

Introducción

Objetivos

- Entender que hace un compilador
- Entender cómo funciona un compilador
- Entender cómo se construye un compilador

Porque estudiar Lenguajes y Compiladores?

1. Aumentar la capacidad de expresión

Adquirir una mayor capacidad para expresar ideas y algoritmos de manera efectiva. Cada lenguaje tiene sus propias características y paradigmas que permiten abordar problemas desde diferentes ángulos.

2. Aprender a construir un sistema grande y confiable

Permite entender mejor cómo se ejecutan los programas, esto incluye el manejo de la memoria, la optimización del código y la gestión de recursos.

Porque estudiar Lenguajes y Compiladores?

3. Mejorar la capacidad de aprender nuevos lenguajes

Estudiar varios lenguajes y los principios de compilación te proporciona una base sólida que facilita el aprendizaje de nuevos lenguajes de programación.

4. Mejorar la comprensión del comportamiento del programa

Implica comprender cómo se diseñan y construyen sistemas grandes y complejos. Aprender sobre modularidad, gestión de dependencias, compilación eficiente y generación de código.

Porque estudiar Lenguajes y Compiladores?

5. Ver muchos conceptos básicos de informática en funcionamiento

Conceptos como estructuras de datos, algoritmos, teoría de autómatas, teoría de lenguajes formales, y más.

Lenguajes de Programación

Intérpretes que ejecutan los programas

Ejecución Inmediata

Portabilidad

Interactividad



Lenguajes de Programación

Compiladores que traducen los programas

Análisis Exhaustivo

Traducción a Código Máquina

Optimización de Código

Ejecución Eficiente



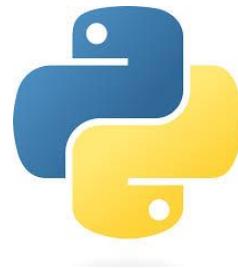
Implementaciones

Los compiladores dominan los lenguajes de bajo nivel



Implementaciones

Los intérpretes dominan los lenguajes de alto nivel



Implementaciones

Algunas implementaciones de lenguaje proporcionan ambos



Tendencia: Intérprete + compilador JIT

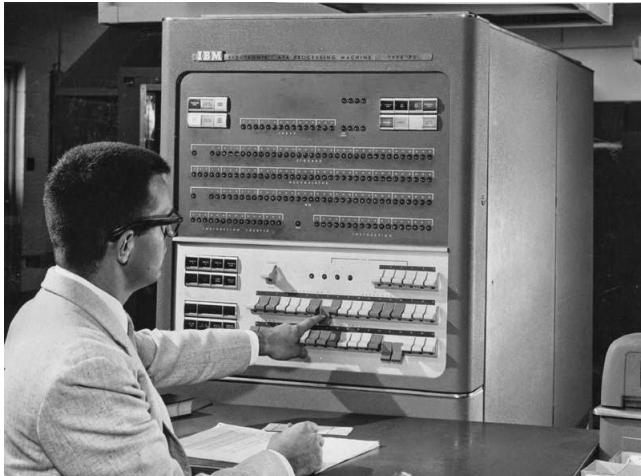
“Abstracción es ignorancia selectiva” - Andrew Koenig

Historia

1954: IBM desarrolla el 704

El costo del software excede al hardware

Toda la programación hecha en ensamblador



Historia

Speedcoding:

- Fue uno de los primeros lenguajes de alto nivel desarrollados para facilitar la programación en computadoras.
- Fue creado por John Backus en 1953 para el IBM 701
- Intérprete
- 10-20 veces más lento que el ensamblador escrito a mano

Fortran I (Formula Translation)

- Creado por John Backus
- Traduce código de alto nivel a ensamblador
- No fue el primer intento pero si el primero exitoso



Fortran I

- 1954-57 - El proyecto FORTRAN I
- 1958 - 50 % de software en el mundo escrito en Fortran
- El tiempo de desarrollo se redujo y el desempeño era equiparable a ensamblador

FOR COMMENT		FORTRAN STATEMENT			IDENTI- FICATION	
STATEMENT NUMBER	CONTINUATION NUMBER	1	2	3	4	5
C		PROGRAM FOR FINDING THE LARGEST VALUE				
C	X	ATTAINED BY A SET OF NUMBERS				
		DIMENSION A(999)				
		FREQUENCY 30(2,1,10), 5(100)				
		READ 1, N, A(I), I = 1,N				
		FORMAT (I3/(12F6.2))				
		BIGA = A(1)				
1		DO 20 I = 2,N				
		30 IF (BIGA >= A(I)) 10,20,20				
		10 BIGA = A(I)				
		20 CONTINUE				
		PRINT 2, N, BIGA				
		2 FORMAT (22H1THE LARGEST OF THESE 13, 12H NUMBERS IS F7.2)				
		STOP 77777				

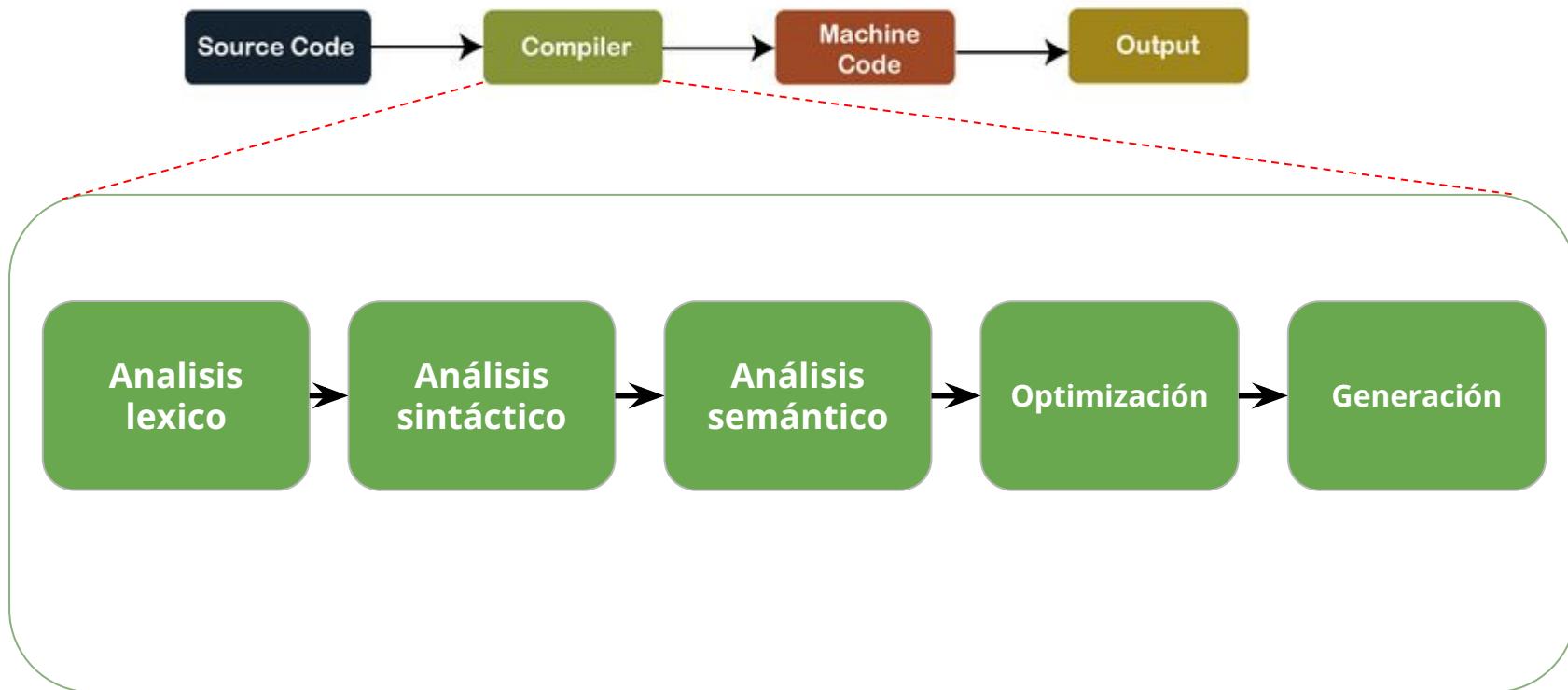
Fortran I

Ampliamente adoptado en la computación científica. Su desarrollo marcó el inicio de la programación de alto nivel y allanó el camino para otros lenguajes más modernos.

FORTRAN han influido en el diseño de muchos otros lenguajes y compiladores modernos. La eficiencia y la optimización para cálculos numéricos, características centrales de FORTRAN.

Compiladores

Estructura de un compilador



Estructura de un compilador

Analisis
lexico

- Convierte el código fuente en una secuencia de palabras (tokens).
- Los tokens son la unidad más pequeña luego de las letras.
- Un token puede representar una palabra clave, un identificador, un operador, un delimitador, etc.
- Cuenta con un alfabeto.

Estructura de un compilador

Analisis
lexico

Primer Paso:
Reconocer palabras

Ejemplo:

Esta es una oración.

Estructura de un compilador

Analisis
lexico

Ejemplo:

taes se anun nociora

Estructura de un compilador

Analisis
lexico

- Los analizadores léxicos dividen los programas en “tokens”:

if x == y then z = 1; else z =2;

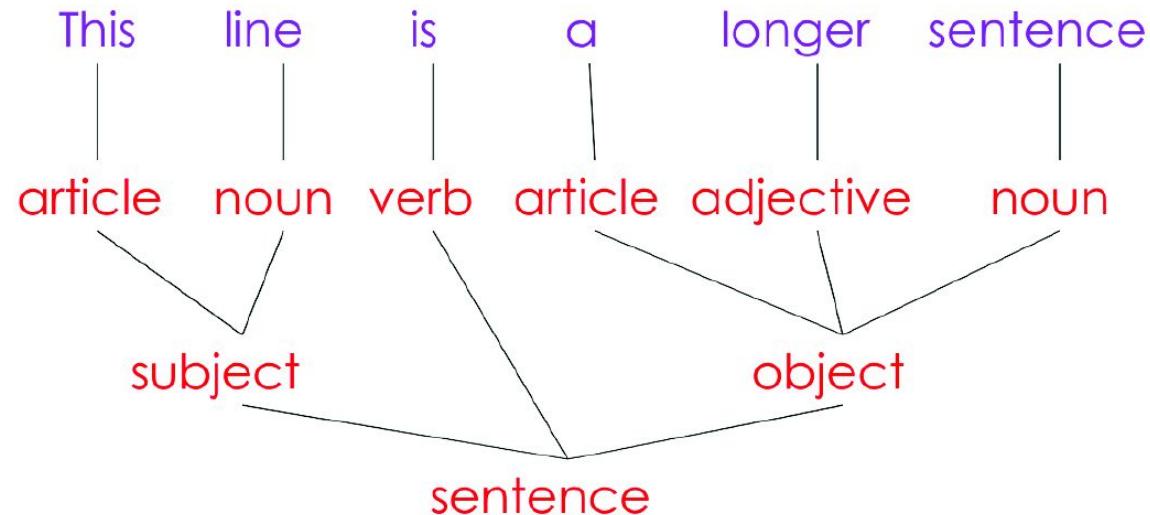
Estructura de un compilador

Análisis
sintáctico

- Entender la estructura de la oración, o sea verifica que la secuencia de tokens siga las reglas de la gramática del lenguaje.
- Parsing = Brindar estructura a una oración
(Generalmente un árbol)
- Detecta errores sintácticos y proporciona mensajes de error informativos para ayudar a los programadores a corregirlos.

Estructura de un compilador

Análisis
sintáctico



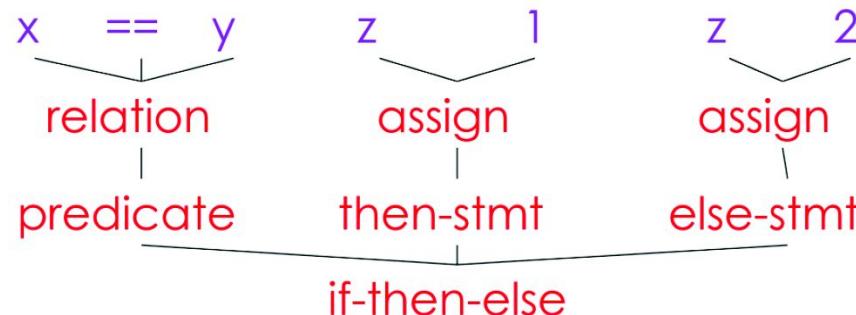
Estructura de un compilador

Análisis
sintáctico

Consideremos

if x == y then z = 1; else z =2;

Diagrama:



Estructura de un compilador

Análisis semántico

- Una vez entendida la estructura de la oración necesitamos entender su significado.
- Detección de inconsistencias.
- Asegura que las variables y funciones se utilicen dentro del alcance en el que fueron definidas.
- Asegura que las operaciones se realicen entre operandos de tipos compatibles.

Estructura de un compilador

Análisis
semántico

Ejemplo:

Juan dijo que José dejó su tarea en casa

¿Quien la dejó en casa?

Estructura de un compilador

Análisis
semántico

Ejemplo:

Juan dijo que Juan dejó su tarea en casa

¿El multiverso de los Juan? ¿Cuantos Juan son?

Estructura de un compilador

Análisis
semántico

Ejemplo:

```
{  
    int juan = 3;  
    {  
        int juan = 4;  
        cout << juan;  
    }  
}
```

Estructura de un compilador

Análisis
semántico

Los compiladores ejecutan diversas verificaciones semánticas, por ejemplo verificación de tipos:

Pikachu es el mejor Digimon

“Type mismatch” entre Pikachu y Digimon

Estructura de un compilador

Optimización

- Las optimizaciones pueden realizarse en varias etapas de la compilación, principalmente en el nivel intermedio (código intermedio) y el nivel del código máquina.

Estructura de un compilador

Optimización

Objetivos de Optimización

- Mejora del Rendimiento
 - Reducir el tiempo de ejecución del programa
 - Minimizar el número de instrucciones ejecutadas
- Reducción del Tamaño del Código
 - Disminuir la cantidad de memoria para almacenar el programa.
 - Compactar el código eliminando instrucciones redundantes.
- Eficiencia en el Uso de Recursos:
 - Optimizar el uso de registros y memoria.
 - Mejorar el rendimiento de la caché.

Estructura de un compilador

Optimización

Ejemplo simple:

$$X = Y * 0$$

..

$$X = 0$$

Estructura de un compilador

Generación

- Al día de hoy muchos compiladores generan representaciones intermedias.
- En cada nivel se reduce la abstracción.
- El código intermedio es una representación abstracta y simplificada del programa fuente que se genera durante el proceso de compilación, sirve como una etapa intermedia que facilita la optimización y la generación de código final para diferentes arquitecturas de hardware.

Estructura de un compilador

Generación

Objetivos del código intermedio

- Independencia de la Máquina
- Facilitación de Optimización
- Simplificación de la Generación de Código (puente entre el código fuente de alto nivel y el código máquina de bajo nivel)



ANÁLISIS LÉXICO

Ing. Max Cerna

Agenda

1. Análisis léxico
2. Ejemplos de análisis léxico
3. Lenguajes regulares
4. Lenguajes formales
5. Especificaciones léxicas

Análisis léxico

¿Qué queremos hacer?

if ($i == j$)

$Z = 0;$

else

$Z = 1;$

realmente es solo una cadena de caracteres:

```
\tif (i == j)\n\t\lz = 0;\n\else\n\t\lz = 1;
```

META: dividir la cadena de entrada en subcadenas

Donde las subcadenas se llaman ***tokens***

Que es un Token?

Una secuencia de caracteres que se **agrupa** y **clasifica** como una unidad.

Clase de Token:

En idioma inglés:

En un lenguaje de programación:

Clase de Token

- **Identificador:** Cadenas de letras o dígitos, comenzando con una letra
- **Número entero:** Una cadena de dígitos no vacía
- **Palabra clave (keyword):** “else” o “if” o “begin” o ...
- **Espacio en blanco:** Una secuencia no vacía de espacios en blanco, saltos de línea y tabulaciones

¿Para qué sirven los Tokens?

Clasificar subcadenas de programas según su rol.

El análisis léxico produce un flujo de tokens que es entrada para el parser.

El analizador se basa en distinciones de tokens

- Un identificador se trata de forma diferente a una palabra clave.

Ejemplo

```
\tif (i == j)\n\t\lz = 0;\n\telse\n\t\lz = 1;
```

Ejercicio

Para el fragmento de código descrito a continuación, seleccione el número correcto de tokens:

x = 0;\n\twhile (x < 10) {\n\t\tx++;}\n}

- a) W = 9; K = 1; I = 3; N = 2; O = 9
- b) W = 11; K = 4; I = 0; N = 2; O = 9
- c) W = 9; K = 4; I = 0; N = 3; O = 9
- d) W = 11; K = 1; I = 3; N = 3; O = 9

Analizador Léxico

Una implementación de analizador léxico debe hacer 2 cosas:

- Reconocer las cadenas que correspondan a los tokens (los lexemas)
- Identificar la categoría de cada lexema

<token class, lexema>

Analizador Léxico

- Normalmente se descarta los tokens "poco interesantes" que no contribuyen al análisis.

Ejemplos: espacios en blanco, comentarios

Analizador Léxico

- Estructura léxica = clases (categorías) de tokens
- Podemos afirmar que un conjunto de cadenas pertenece a una clase de tokens

Ejemplos de análisis léxico

Ejemplo Analizador Léxico

Regla de FORTRAN: Los espacios en blanco son insignificantes
VAR1 es lo mismo que VA R1

Ejemplo Analizador Léxico

Regla de FORTRAN: Los espacios en blanco son insignificantes
VAR1 es lo mismo que VA R1

Terrible diseño

Ejemplo Analizador Léxico

DO 5 I = 1,25

DO 5 I = 1.25

Ejemplo Analizador Léxico

- El objetivo es dividir la cadena. Esto se implementa leyendo de izquierda a derecha, reconociendo un token a la vez.
- Es posible que se requiera "mirar hacia adelante (lookahead)" para decidir dónde termina un token y comienza el siguiente token

Ejemplo Analizador Léxico

if (i == j)

Z = 0;

else

Z = 1;

Ejemplo Analizador Léxico

PL/I: Las palabras clave (keywords) no están reservadas

IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN

Ejemplo Analizador Léxico

PL/I: Las palabras clave (keywords) no estan reservadas

DECLARE (ARG1, . . . , ARGN)

¿Es DECLARE una palabra clave o un identificador de arreglo?

Ejemplo Analizador Léxico

C++ template

 Foo<Bar>

C++ stream

 cin >> var;

Lenguajes Regulares

Lenguaje

Sea el alfabeto Σ un conjunto de caracteres.

Un lenguaje sobre Σ es un conjunto de cadenas de caracteres extraídos de Σ

Notación

- Los lenguajes son conjuntos de cadenas.
- Necesitamos alguna notación para especificar qué conjuntos queremos.
- La notación estándar para los lenguajes regulares es expresiones regulares.

Lenguajes Regulares

Existen varios formalismos para especificar tokens.

Los lenguajes regulares son los más populares.

- Teoría simple y útil.
- Fácil de comprender
- Implementaciones eficientes

Notación de Expresiones Regulares

- **Epsilon**

$$\varepsilon = \{\text{'''}\}$$

Representa una cadena de longitud cero.

No es un conjunto vacío.

Notación de Expresiones Regulares

- **Carácter simple**

'c' = {"c"}

- **Unión**

A + B = {s | s ∈ A or s ∈ B }

Notación de Expresiones Regulares

- Concatenación

$$AB = \{ab \mid a \in A \text{ and } b \in B\}$$

- Iteración

$$A^* = \bigcup_{i \geq 0} A_i \text{ donde } A_i \text{ es } AA\dots A \text{ } i \text{ veces}$$

Expresiones Regulares

Definición

Las expresiones regulares sobre Σ son el conjunto más pequeño de expresiones que incluyen

ϵ

'c' donde $c \in \Sigma$

$A+B$ donde A,B es una rexp sobre Σ

AB donde A,B es una rexp sobre Σ

A^* donde A es una rexp sobre Σ

Ejercicio

Elija los lenguajes regulares que sea equivalentes al lenguaje:

$(0 + 1)^* 1 (0 + 1)^*$ para $\Sigma = 0, 1$

- a) $(01 + 11)^* (0 + 1)^*$
- b) $(0 + 1)^* (10 + 11 + 1) (0 + 1)^*$
- c) $(1 + 0)^* 1 (1 + 0)^*$
- d) $(0+ 1)^* (0 + 1) (0 + 1)^*$

Lenguajes Formales

Lenguajes Formales

- Alfabeto = Caracteres en español
- Lenguaje = Oraciones en español
- Alfabeto = ASCII
- Lenguaje = Programas en C

Sintaxis y Semántica

La notación hasta ahora era imprecisa

$$AB = \{ab \mid a \in A \text{ and } b \in B\}$$

B como una pieza de sintaxis

B como un conjunto
(la semántica de la sintaxis)

Sintaxis y Semántica

La función de significado L asigna la sintaxis a la semántica

$$\varepsilon = \{'''\}$$

$$'c' = \{"c"\}$$

$$A + B = \{ A \cup B \}$$

$$AB = \{ab \mid a \in A \text{ and } b \in B\}$$

$$A^* = \bigcup_{i \geq 0} A^i$$

Sintaxis y Semántica

La función de significado L asigna la sintaxis a la semántica

$$L(\varepsilon) = \{'''\}$$

$$L('c') = \{"c"\}$$

$$L(A + B) = \{ L(A) \cup L(B) \}$$

$$L(AB) = \{ab \mid a \in L(A) \text{ and } b \in L(B)\}$$

$$L(A^*) = \bigcup_{i \geq 0} L(A)^i$$

LENGUAJES FORMALES

¿Por qué usar una función de significado?

- Dejar claro qué es la sintaxis, qué es la semántica.
- Permite considerar la notación como un tema separado
- Porque las expresiones y significados no son 1-1*

LENGUAJES FORMALES

*El significado es muchos a uno: ¡nunca uno a muchos!

- Indica que una expresión sintáctica específica no puede tener múltiples significados semánticos diferentes.
- Esto sería una ambigüedad semántica y podría llevar a problemas de interpretación o ejecución en el lenguaje.

Ejemplo: **x = y**

no puede significar dos operaciones diferentes al mismo tiempo (ej: asignación y comparación).

Especificaciones léxicas

Expresiones Regulares

Se pueden describir ***tokens*** usando ***expresiones regulares***

Ejemplos

Token - Keywords

Abreviación: 'E' 'L' 'S' 'E' se puede escribir como 'ELSE'

keyword: “ELSE” or “IF” or “BEGIN” or ...

'ELSE' + 'IF' + 'BEGIN' + ...

Token - Integer

Integer: una cadena no vacía de dígitos

digit = '0' + '1' + '2' + '3' + '4' + '5' + '6' + '7' + '8' + '9'

Abreviación: [0-3] = '0' + '1' + '2' + '3'

digit = [0-9]

integer = digit digit*

Abreviación: digit⁺ = digit digit*

Token - Identifier

Identifier: cadena de letras o números que inician con letra.

letter = 'a' + ... + 'z' + 'A' + ... + 'Z'

Abreviación: [a-z] = 'a' + ... + 'z'

identifier = letter (letter + digit)*

Token - Whitespace

Whitespace: cadena de no vacía de espacios en blanco, cambio de línea y tabulaciones.

`whitespace = (' ' + '\n' + '\t')+`

Ejemplo - Mail Address

ej: estudiante@url.edu.gt

$$\Sigma = \text{letras} \cup \{ '.', '@' \}$$

$$\text{name} = \text{letter}^+$$

$$\text{address} = \text{name} '@' \text{name} '.' \text{name} ('.' \text{name} + \varepsilon)$$

Ejemplo - Phone Number

ej: +502-2341-5678

$$\Sigma = \text{dígitos} \cup \{-, +\}$$

$$\text{section} = \text{digit}^4$$

$$\text{area} = '+'\text{digit}^3$$

$$\text{phone} = (\text{area}'-'+' + \varepsilon) \text{section}'-' \text{section}$$

Ejemplo- Número decimal formato Pascal

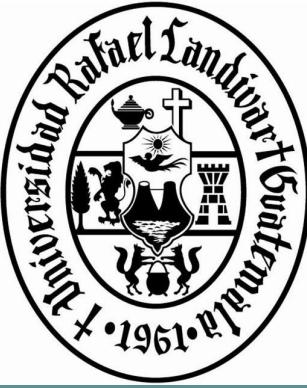
digit = [0-9]

digits = digit⁺

opt_fraction = ('.' digits) + ε

opt_exponent = ('E' ('+' '+' '-' '+') digits) + ε

num = digits opt_fraction opt_exponent



AUTÓMATAS

Regex convertido en robot

Ing. Max Cerna





Tips de software

- Mantenlo simple
- No optimice prematuramente
- Diseñe sistemas que puedan probarse
- Es más fácil modificar un sistema que funciona que hacer que un sistema funcione



Agenda

1. Especificación léxica
2. Autómata finito
3. Expresiones hacia autómatas
4. NFA hacia DFA
5. Implementación de un autómata finito



Objetivo

Convertir expresiones regulares en autómatas finitos



Especificación Léxica



Especificación Léxica

Al menos uno	AA^*	A^+
Unión	$A+B$	$A B$
Opción	$A+\varepsilon$	$A?$
Rango	$'a'+'b'+'...'+'z'$	$[a - z]$
Rango (exclusión) - Complemento de $[a - z]$		$[\neg a - z]$



Especificación Léxica

¿Como saber si?

$$s \in L(R)$$

expresión regular

cadena

lenguaje

The diagram shows the expression $s \in L(R)$. An arrow points from the label "expresión regular" above the letter R to the letter R itself. Another arrow points from the label "cadena" below the letter s to the letter s. A brace under the expression $L(R)$ has an arrow pointing to the label "lenguaje" below it.



Especificación Léxica

¿Como saber si?

$$s \in L(R)$$

Una respuesta de sí o no, no es suficiente

- dividir la entrada en tokens
- Adaptar las expresiones regulares a este objetivo.



Especificación Léxica

- 1) Escribir una regex para los lexemas de cada clase (categoría) de token
 - Número = digit+
 - Palabra clave = 'if' + 'else' + ...
 - Identificador= letter (letter + digit)*
 - OpenPar = '('
 - ...



Especificación Léxica

- 2) Construir R, coincidiendo con todos los lexemas para todos los tokens

$R = \text{Keyword} + \text{Identifier} + \text{Number} + \dots$

$= R_1 + R_2 + \dots$

R debe ser capaz de coincidir con cualquier secuencia de caracteres que forme un lexema válido para cualquier tipo de token que estemos interesados en reconocer.

Combinamos expresiones regulares más pequeñas que representan los diferentes tipos de tokens.



Especificación Léxica

- 3) Considerando la entrada $x_1 \dots x_n$

Para $1 \leq i \leq n$ verificar

$$x_1 \dots x_i \in L(R)$$

- 4) Si éxito, entonces sabemos que:

$$x_1 \dots x_n \in L(R_j) \text{ para algún } j$$

- 5) Eliminar $x_1 \dots x_i$ de la entrada e ir a (3)



Especificación Léxica

¿Es utilizada toda la entrada?



Especificación Léxica

¿Cual token se debe utilizar?



Especificación Léxica

¿Y si la cadena no coincide?



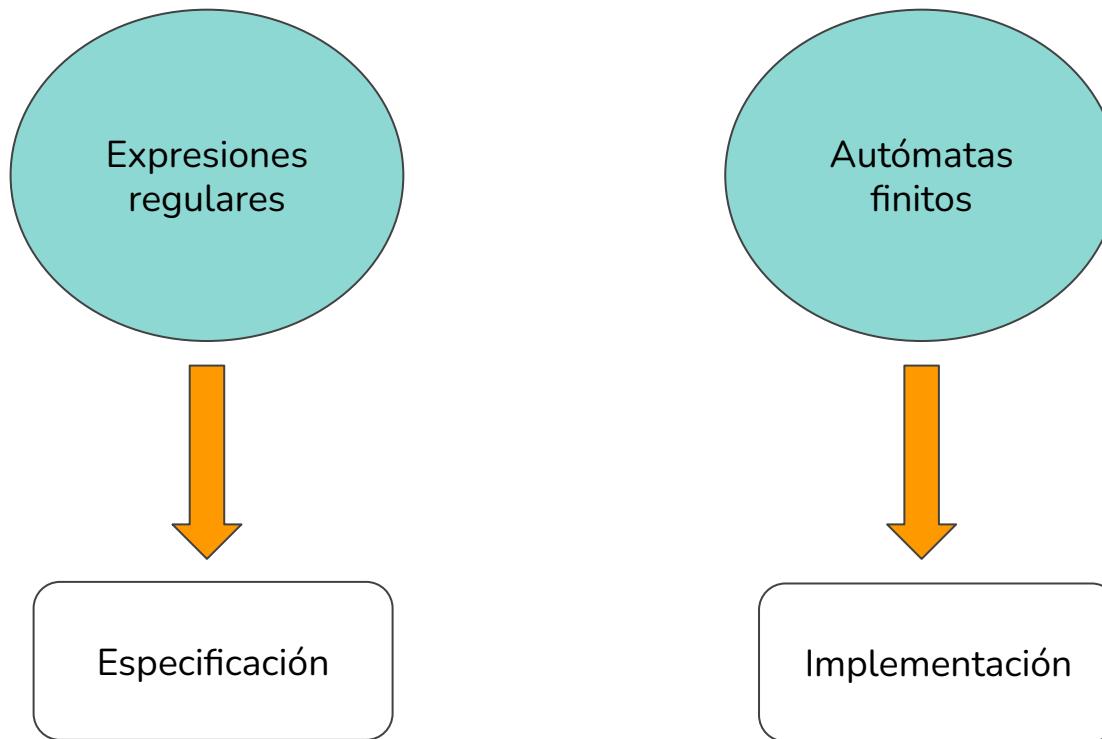
Resumen

- Las expresiones regulares son una notación concisa para patrones de cadenas.
- El uso en análisis léxico requiere pequeñas extensiones.
 - Para resolver ambigüedades
 - Para manejar errores
- Buenos algoritmos conocidos
 - Requerir solo una pasada sobre la entrada
 - Pocas operaciones por carácter (búsqueda de tabla)

Autómata finito



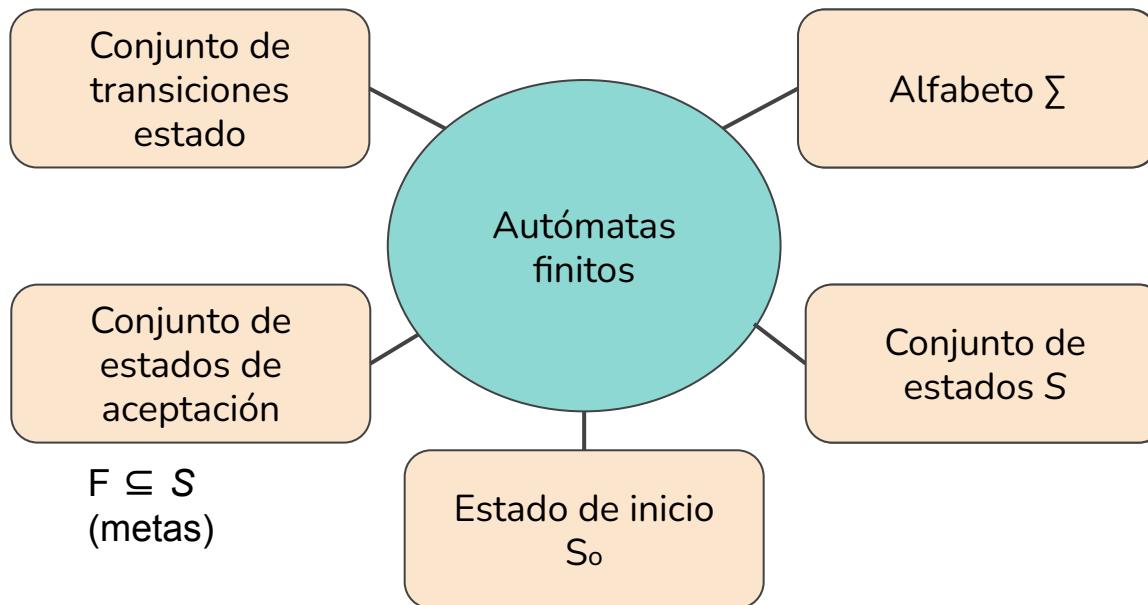
Autómata finito





Autómata finito

estado →_{entrada} estado





Autómata finito

Transición

$$s_1 \xrightarrow{a} s_2$$

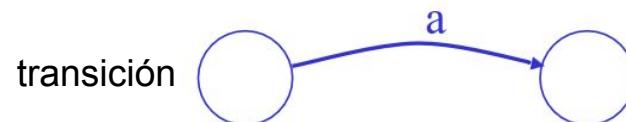
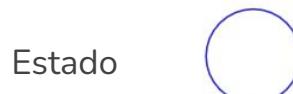
Se interpreta como:

“En el estado s_1 con la entrada a dirigirse hacia s_2 ”

- Si finaliza la entrada y está en estado de aceptación entonces **aceptar**
- De lo contrario entonces **rechazar**



Autómata finito - Notación



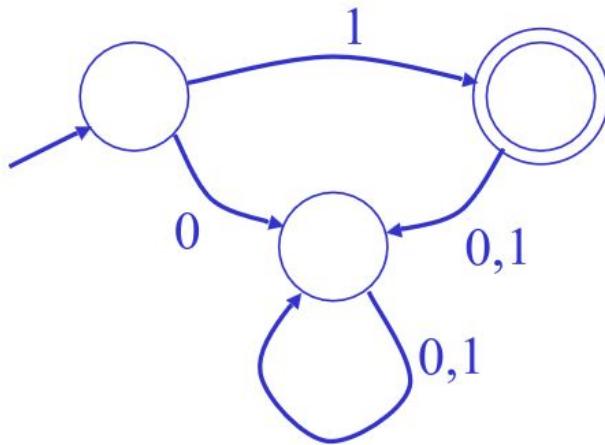


Autómata finito - Ejemplo

Un autómata finito que solo
acepta “1“



Autómata finito - Ejemplo





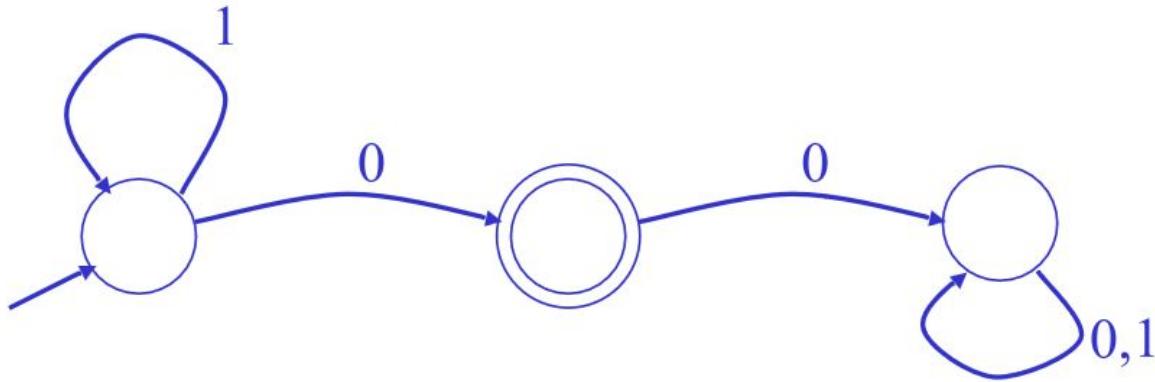
Autómata finito - Ejemplo

Un autómata finito que acepta
cualquier cantidad de "1"
seguido de un 0 simple.

Alfabeto: 0,1



Autómata finito - Ejemplo





Autómata finito - Ejemplo

Seleccione el lenguaje regular que denota el mismo lenguaje que este autómata finito:

- $(0 + 1)^*$
- $(1 * + 0)(1 + 0)$
- $1 * + (01) * + (001) * + (000 * 1) *$
- $(0 + 1) * 00$

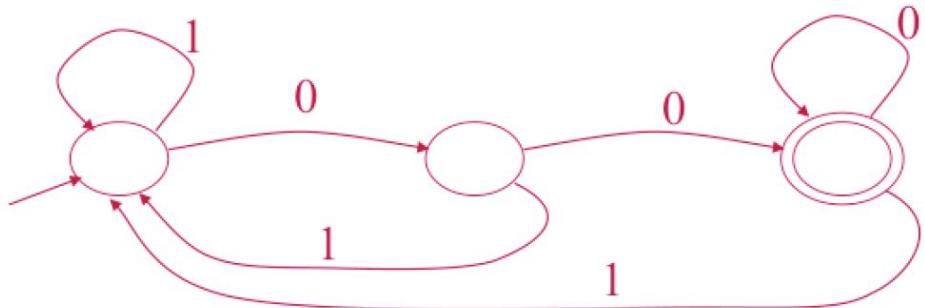
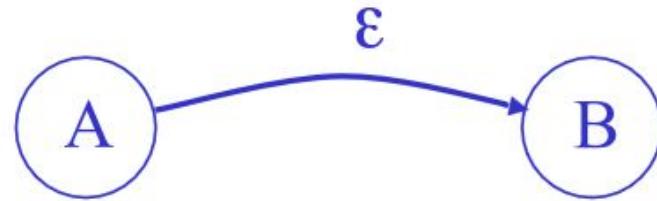


Figura: Automata Finito



Autómata finito - Epsilon



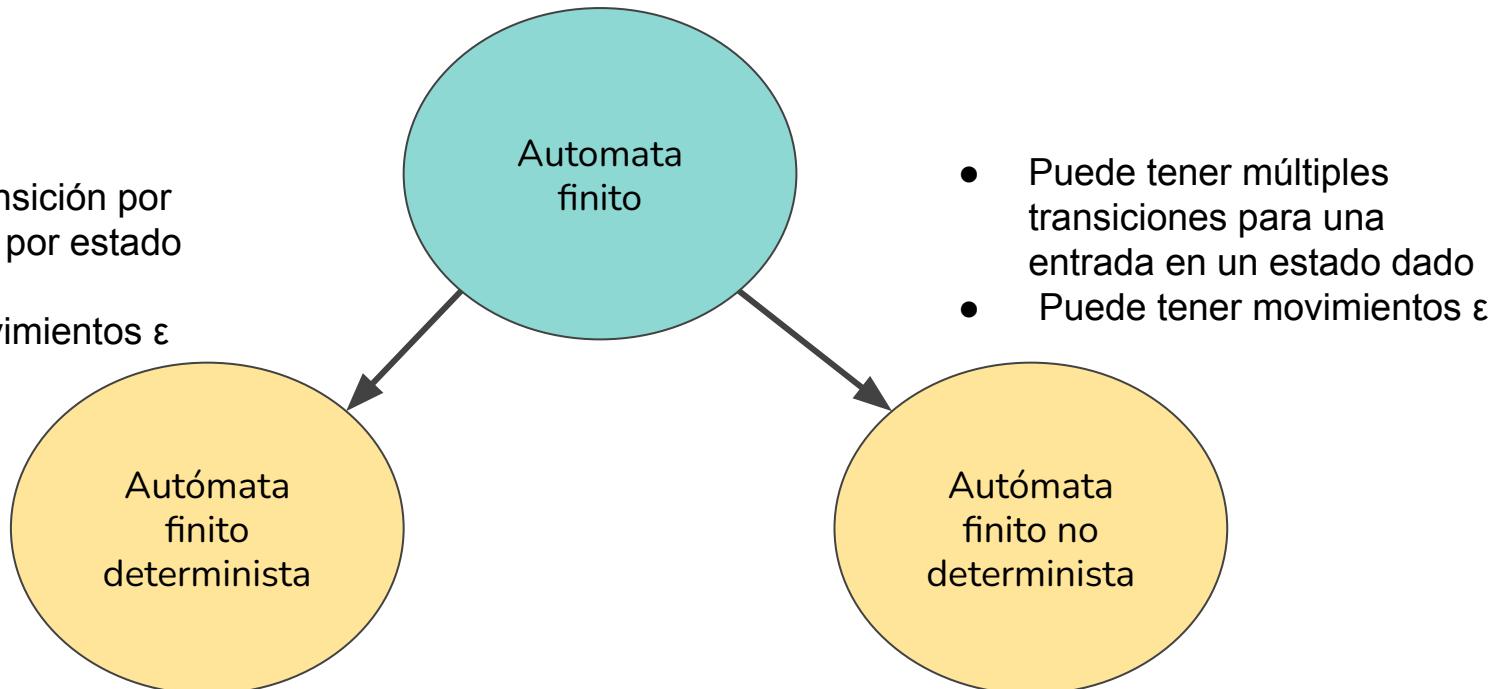
La máquina puede pasar del estado A al estado B, sin leer la entrada

Sólo existen en NFA



Autómata finito

- Una transición por entrada por estado
- Sin movimientos ϵ





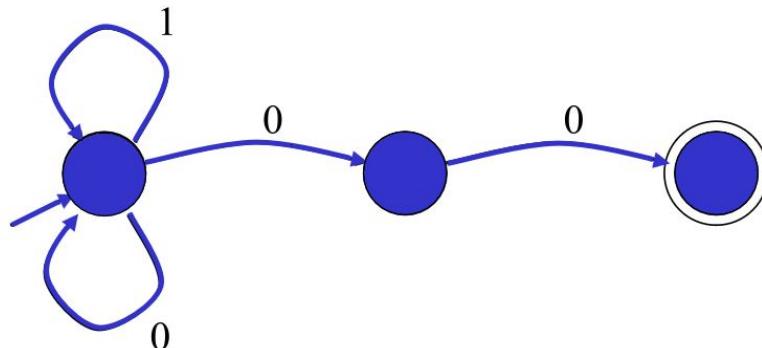
Autómata finito

- Un DFA toma solo un camino a través del gráfico de estado
- Un NFA puede elegir



Autómata finito

- Un NFA puede desplazarse hacia varios estados



Input: 1 0 0



NFA vs DFA

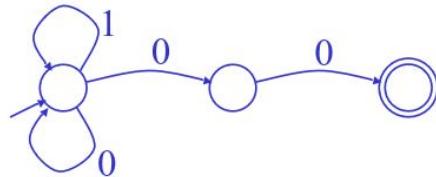
- NFA y DFA reconocen el mismo conjunto de idiomas regulares
- Los DFA son más rápidos de ejecutar, no hay opciones a considerar.



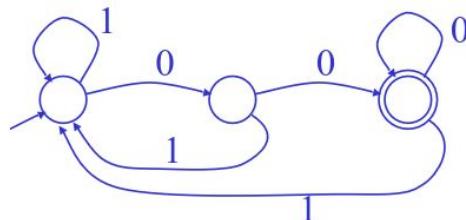
NFA vs DFA

- Para un idioma determinado, NFA puede ser más simple que DFA

NFA



DFA





Expresiones hacia Autómatas

Para cada regex, definir un NFA

Notación: NFA para regex M



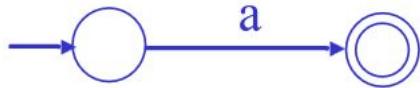


Expresiones hacia Autómatas

Para ϵ



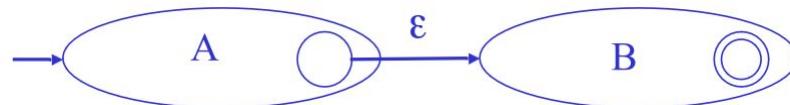
Para la entrada a



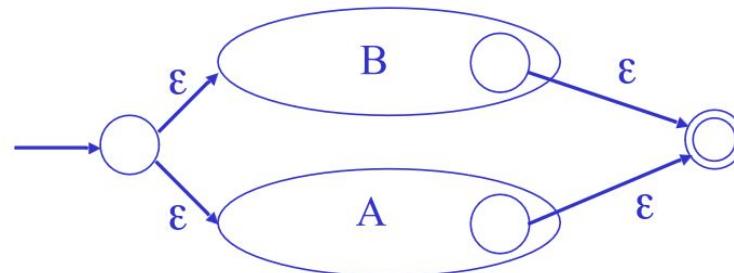


Expresiones hacia Autómatas

Para AB



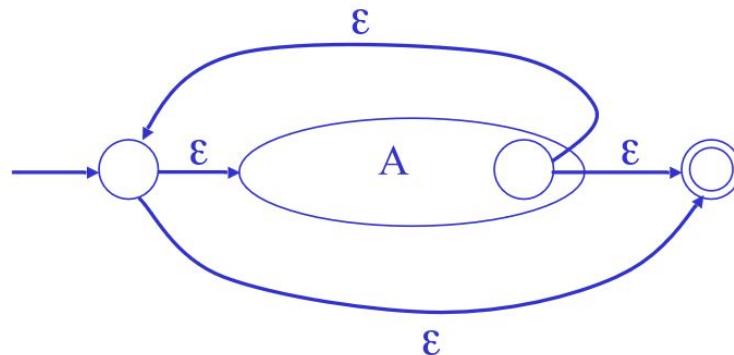
Para A+B





Expresiones hacia Autómatas

Para A^*



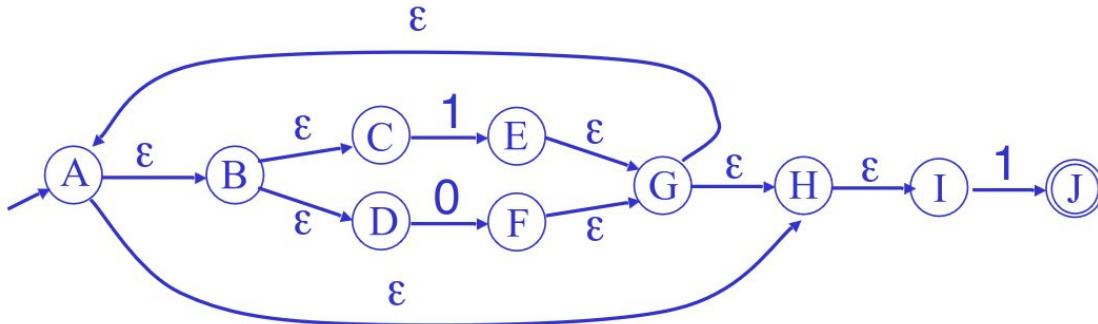


Expresiones hacia Autómatas

Considere la expresión regular $(1+0)^*1$



Expresiones hacia Autómatas



NFA hacia DFA



NFA a DFA: El Truco

- Construcción del NFA
- Cada estado del DFA = un subconjunto no vacío de estados del NFA
- Estado inicial = el conjunto de estados del NFA accesibles a través de movimientos ϵ desde el estado inicial del NFA

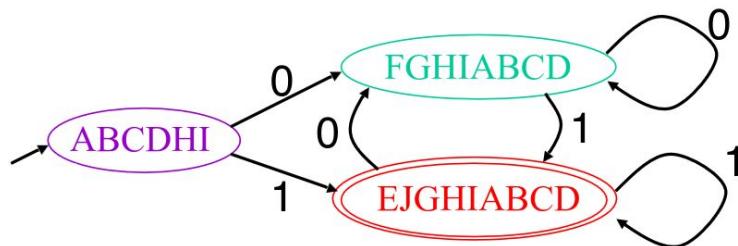
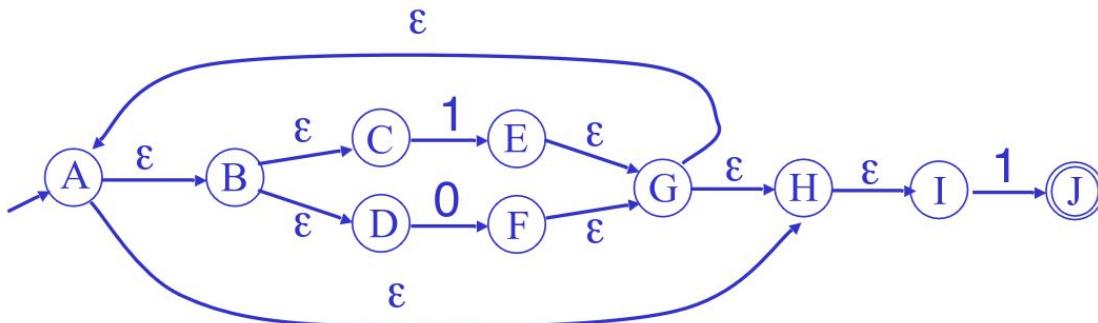


NFA HACIA DFA

- Un NFA puede estar en varios estados en cualquier momento
- ¿Cuantos?

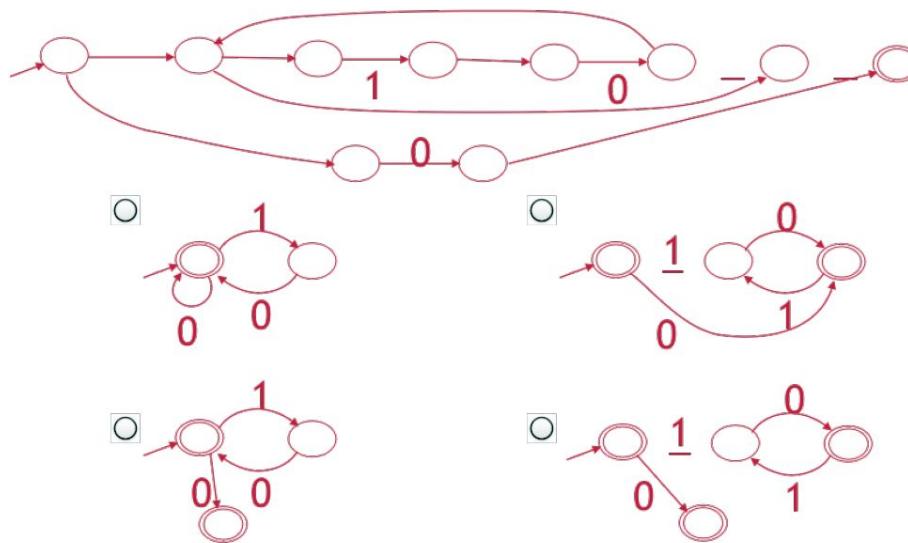


NFA HACIA DFA





NFA HACIA DFA



Implementación de un autómata finito



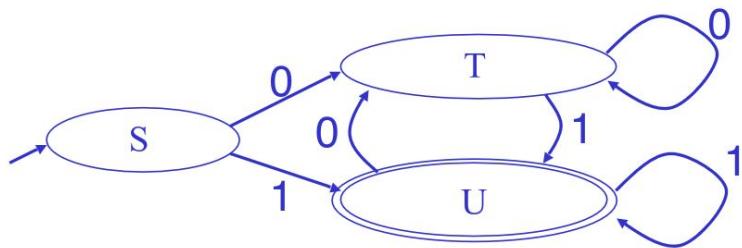
Implementación de un autómata finito

Un DFA puede ser implementado mediante una tabla “T” 2D

- Una dimensión son los "estados"
- Otra dimensión es el símbolo de entrada
- Para cada transición $S_i \rightarrow a S_k$ define $T[i, a] = k$



Implementación de un autómata finito



	0	1
S	T	U
T	T	U
U	T	U



Implementación de un autómata finito

La conversión de NFA a DFA es el núcleo de herramientas como flex.

Sin embargo, los DFA pueden ser muy grandes.

En la práctica, las herramientas tipo flex sacrifican velocidad por espacio en la elección de las representaciones de NFA y DFA.

Ejemplos

Conversión NFA a DFA





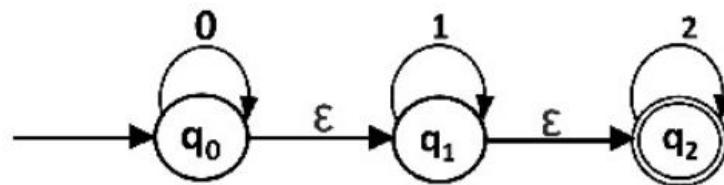
ϵ -closure (cerradura)

Es el conjunto de estados accesibles desde un conjunto dado sin consumir ningún símbolo de la cadena de entrada.

Estado Inicial: La conversión comienza tomando el ϵ -closure del estado inicial del NFA. Este ϵ -closure se convierte en el estado inicial del DFA.



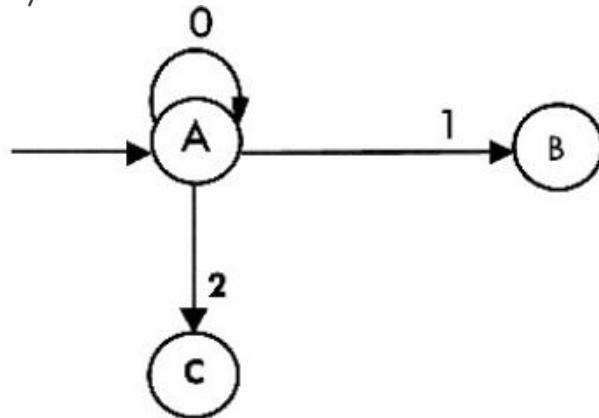
Ejemplo 1





Ejemplo 1

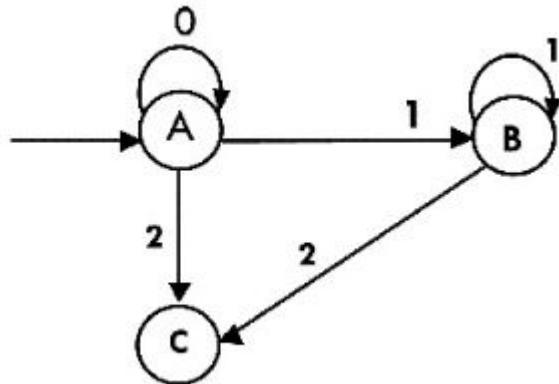
Después de analizar A para 0,1 y 2





Ejemplo 1

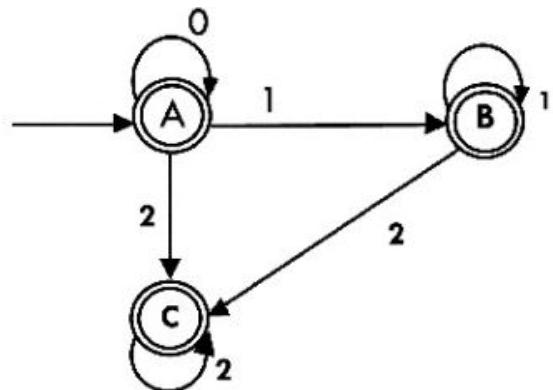
Después de analizar B para 0,1 y 2





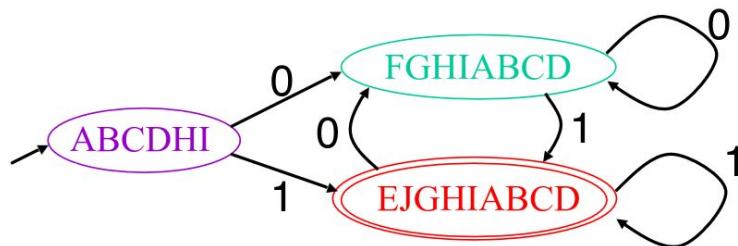
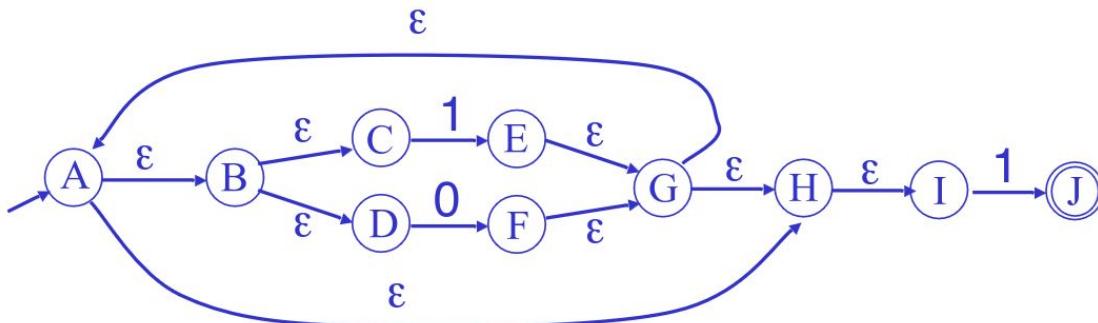
Ejemplo 1

Después de analizar C para 0,1 y 2



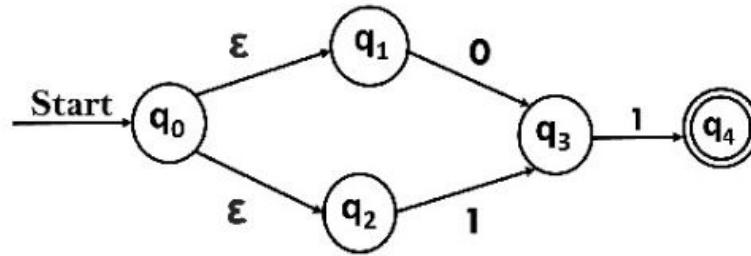


NFA HACIA DFA





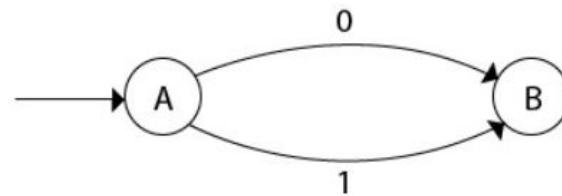
Ejemplo 2





Ejemplo 2

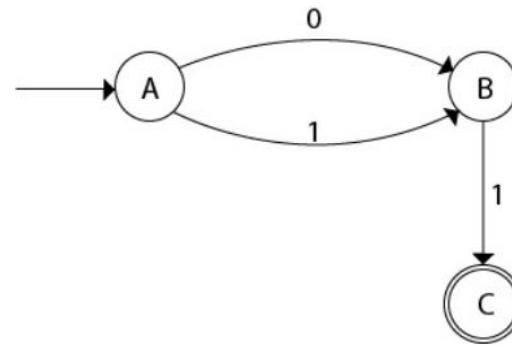
Después de analizar A para 0,1

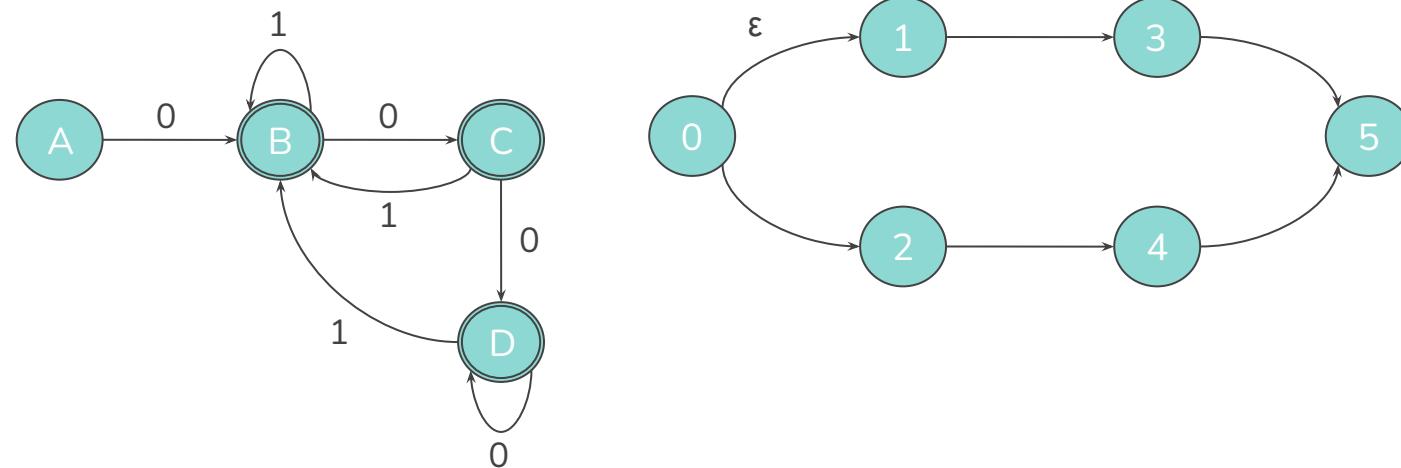




Ejemplo 2

Después de analizar B para 0,1

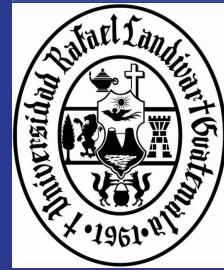




PARSING

Regex convertido en robot

Ing. Max Cerna



Agenda

1. Parsers
2. Gramáticas libres de contexto (CFG)
3. Derivaciones
4. Ambigüedad

Parsers

PARSING

Lenguajes regulares

- Los lenguajes formales más débiles que se pueden utilizar
 - Son capaces de describir patrones simples como palabras clave, identificadores, y ciertas estructuras repetitivas en el código fuente
- Muchas aplicaciones
 - Validación de Patrones Simples
 - Filtros y Procesadores de Texto

PARSING

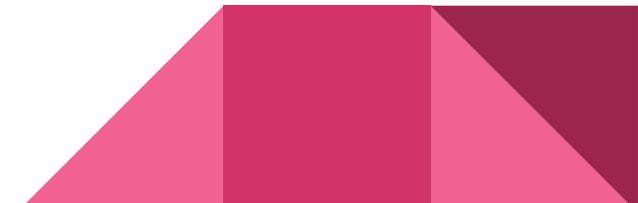
Muchos lenguajes no son regulares

Las cadenas de paréntesis balanceados no son regulares

$$(i)^i \mid i \geq 0$$

¿Qué pueden expresar los lenguajes regulares?

- Expresan propiedades que dependen de contar hasta cierto punto
- Tienen una capacidad limitada de "memoria"
- Debido a esta limitación, no pueden manejar lenguajes que requieran un conteo exacto de elementos



PARSING

INPUT

Secuencia de tokens del lexer

OUTPUT

Árbol de parsing del programa

PARSING



PARSING

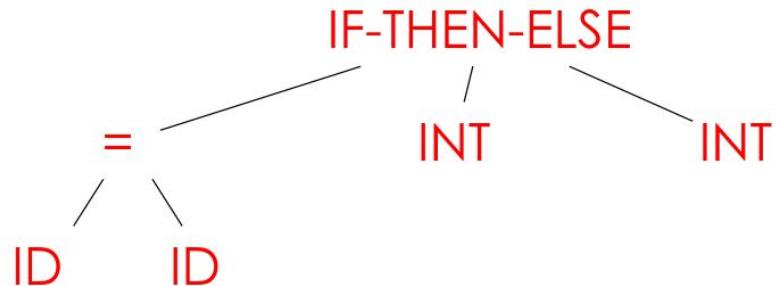
Cool (pseudo lenguaje)

if x = y then 1 else 2 fi

Entrada al parser (cadena de tokens)

IF ID = ID THEN INT ELSE INT FI

Salida del parser



Gramáticas libres de contexto (CFG)

CFG

- No todas las cadenas de tokens son programas. . .
- . . . el analizador debe distinguir entre cadenas de tokens válidas e inválidas
- Nosotros necesitamos
 - Un lenguaje para describir cadenas válidas de tokens
 - Un método para distinguir cadenas de tokens válidas de las no válidas

CFG

- Los lenguajes de programación tienen estructura recursiva
- Una EXPR es

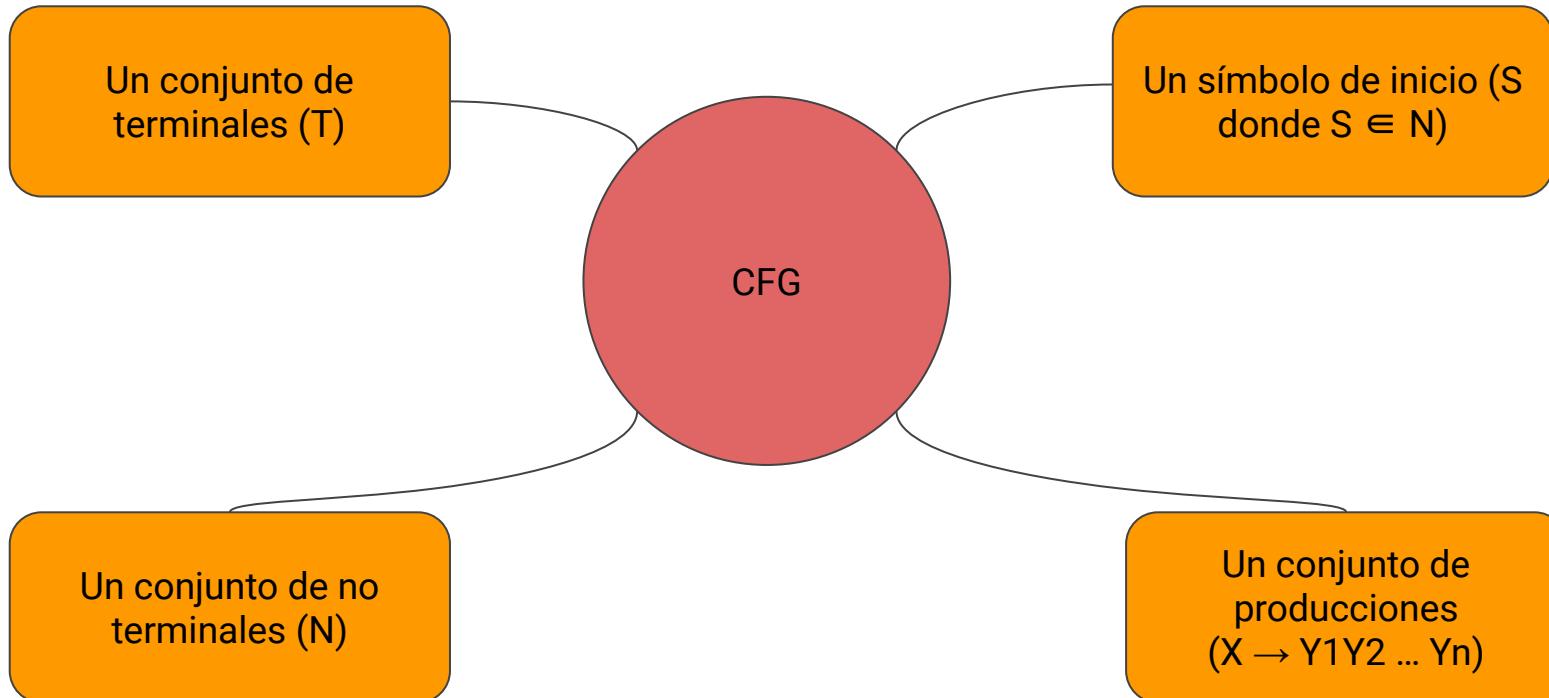
IF EXPR then EXPR else EXPR fi

while EXPR loop EXPR pool

...

- Las gramáticas libres de contexto son una notación natural para esta estructura recursiva

CFG consta de..



CFG

Las producciones se pueden leer como reglas

CFG - Notación

- Los no terminales se escriben en mayúsculas.
- Los terminales se escriben en minúsculas.
- El símbolo de inicio es el lado izquierdo de la primera producción.

CFG - Algoritmo

- 1) Comience con una cadena con solo el símbolo de inicio S
- 2) Reemplace cualquier X no terminal en la cadena por el lado derecho de alguna producción ej: $X \rightarrow Y_1Y_n$
- 3) Repita (2) hasta que no haya no terminales

CFG

- Los terminales se llaman así porque no hay reglas para reemplazarlos
- Una vez generados, los terminales son permanentes
- Los terminales deben ser tokens del idioma.

CFG - EXPRESIONES ARITMÉTICAS SIMPLES

$E \rightarrow E * E$

| $E + E$

| (E)

| id

CFG - Ejemplo

Cual de las siguientes cadenas están en el lenguaje dado por la CFG

- abcba
- acca
- aba
- abcbcba

$$S \rightarrow aXa$$

$$X \rightarrow \epsilon$$

$$| bY$$

$$Y \rightarrow \epsilon$$

$$| cXc$$

CFG

- Permite determinar si una cadena pertenece al lenguaje definido por la gramática.
- Además de verificar si una cadena pertenece al lenguaje, es esencial generar un árbol de análisis sintáctico (parse tree).
- Debe manejar los errores con gracia.
- Necesita una implementación de CFG (p. ej., Bison, CUP).
- La forma de la gramática es importante
 - Muchas gramáticas generan el mismo lenguaje
 - Las herramientas son sensibles a la gramática.

Derivaciones

Derivaciones

Una derivación es una secuencia de producciones:

$S \rightarrow \dots \rightarrow \dots \rightarrow \dots \rightarrow \dots \rightarrow \dots$

Una derivación puede dibujarse como un árbol

- El símbolo de inicio es la raíz del árbol
- Para una producción $X \rightarrow Y_1\dots Y_n$ agregar los hijos $Y_1\dots Y_n$ al nodo

Derivaciones

Gramática $E \rightarrow E + E \quad | \quad E * E \quad | \quad (E) \quad | \quad id$

Cadena $id * id + id$

Derivaciones

E

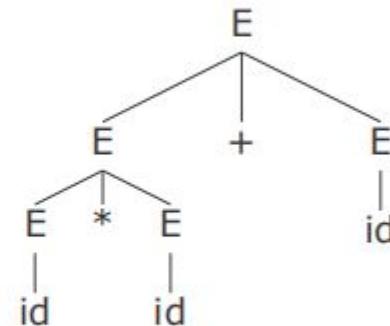
$\rightarrow E + E$

$\rightarrow E * E + E$

$\rightarrow id * E + E$

$\rightarrow id * id + E$

$\rightarrow id * id + id$



Derivaciones

- Un árbol de análisis tiene
 - Terminales en las hojas
 - No terminales en los nodos interiores
- Un recorrido en orden de las hojas es la entrada original
- El árbol de análisis muestra la asociación de operaciones, la cadena de entrada no

Derivaciones

El ejemplo es una derivación por la izquierda (left-most derivation).

En cada paso, reemplaza el no terminal más a la izquierda.

Existe una noción equivalente de una derivación por la derecha (right-most derivation).

E

$\rightarrow E + E$

$\rightarrow E + id$

$\rightarrow E * E + id$

$\rightarrow E * id + id$

$\rightarrow id * id + id$



Derivaciones

E

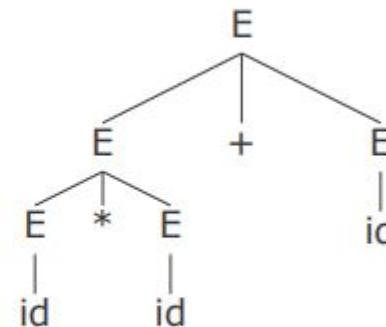
$\rightarrow E + E$

$\rightarrow E + id$

$\rightarrow E * E + id$

$\rightarrow E * id + id$

$\rightarrow id * id + id$



Derivaciones

Tenga en cuenta que las derivaciones por la derecha y por la izquierda tienen el mismo árbol de análisis

Derivaciones

¿Cuál de las siguientes es una derivación válida de la gramática dada?

$$S \rightarrow aXa$$

$$X \rightarrow \epsilon \quad | \quad bY$$

$$Y \rightarrow \epsilon \quad | \quad cXc \quad | \quad d$$

- 1) S
aXa
abYa
acXca
acca

- 2) S
aa

- 3) S
aXa
abYa
abcXca
abcbYca
abcbdca

- 4) S
aXa
abYa
abcXcda
abccda

Derivaciones

- No solo estamos interesados en si $s \in L(G)$. Necesitamos un árbol de análisis para s
- Una derivación define un árbol de análisis. Pero un árbol de análisis puede tener muchas derivaciones
- Las derivaciones más a la izquierda y más a la derecha son importantes en la implementación del analizador

Ambigüedad

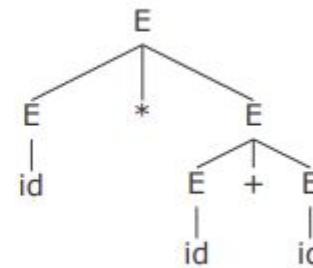
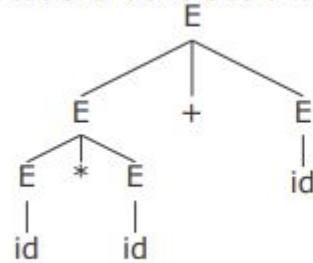
Ambigüedad

Gramática $E \rightarrow E + E \quad | \quad E * E \quad | \quad (E) \quad | \quad id$

Cadena $id * id + id$

Ambigüedad

La cadena tiene dos árboles diferentes



Ambigüedad

- Una gramática es ambigua si tiene más de un árbol de análisis para alguna cadena.
 - De manera equivalente, existe más de una derivación más a la derecha o más a la izquierda para alguna cadena.
- La ambigüedad es MALA
 - Deja indefinido el significado de algunos programas.

Ambigüedad

¿Cuáles de las siguientes gramáticas son ambiguas?

- $S \rightarrow SS \mid a \mid b$
- $E \rightarrow E + E \mid id$
- $S \rightarrow Sa \mid Sb$
- $E \rightarrow E \mid E + E$
- $E \rightarrow -E \mid id$

Ambigüedad

- Hay varias formas de manejar la ambigüedad
- El método más directo es reescribir la gramática inequívocamente, es decir sin ambigüedad

$$E \rightarrow E + E | E$$
$$E \rightarrow id * E | id | (E) * E | (E)$$

Precedencia de * sobre +

Ambigüedad

Considere la gramática

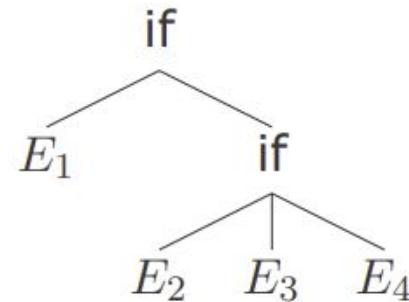
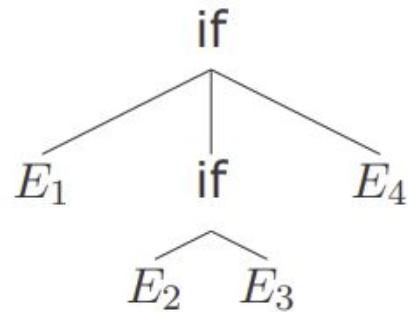
$E \rightarrow \text{if } E \text{ then } E$

| $\text{if } E \text{ then } E \text{ else } E$

| OTHER

Ambigüedad

La expresión `if E1 then if E2 then E3 else E4` tiene dos árboles



Ambigüedad

`else` coincide con el `then` aún sin coincidir más cercano:

`E` → `MIF`

| `UIF`

`MIF` → `if E then MIF else MIF`

| `OTHER`

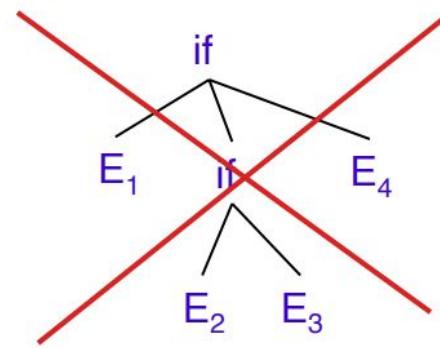
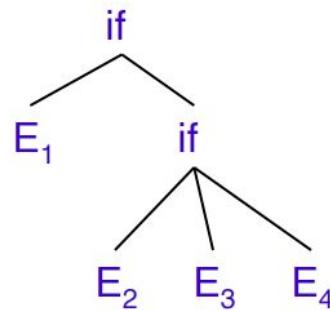
`UIF` → `if E then E`

| `if E then MIF else UIF`

Ambigüedad

Entonces expresión if E1 then if E2 then E3 else E4

UIF → if E then E
I if E then MIF else UIF



Ambigüedad

- No hay técnicas generales para manejar la ambigüedad
- Es imposible convertir automáticamente una gramática ambigua en una no ambigua
- Usada con cuidado, la ambigüedad puede simplificar la gramática
 - A veces permite definiciones más naturales
 - Necesitamos mecanismos de desambiguación

Ambigüedad

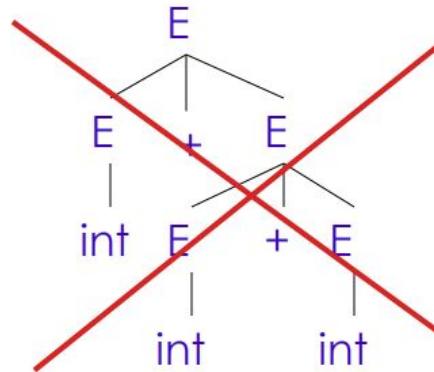
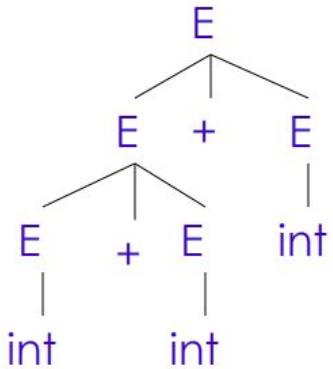
- En lugar de reescribir la gramática:
 - Utiliza la gramática más natural (ambigua)
 - Junto con declaraciones de desambiguación
- La mayoría de las herramientas permiten declaraciones de precedencia y asociatividad para desambiguar las gramáticas



Ambigüedad

Considere la gramática $E \rightarrow E + E | int$

Dos árboles ambiguos para $int + int + int$:

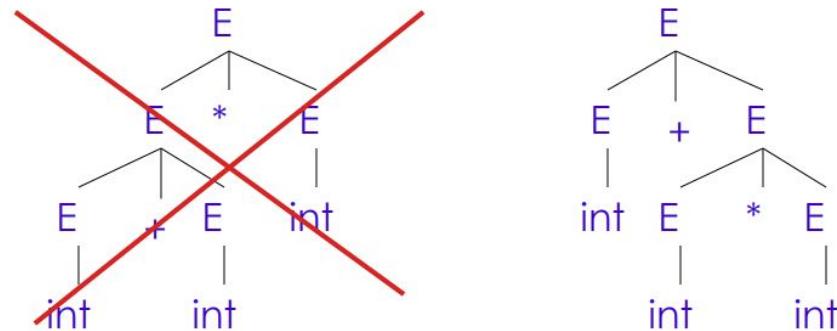


Asociación y declaración por la izquierda: %left +

Ambigüedad

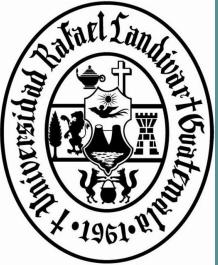
Considere la gramática $E \rightarrow E + E | E * E | \text{int}$

Dos árboles ambiguos para $\text{int} + \text{int} * \text{int}$:



Asociación de precedencia: %left +

%left *



Traducción dirigida por sintaxis

Ing. Max Cerna





Agenda

1. Gestión de errores
2. AST
3. Recursive Descent Parsing
4. Recursive Descent Algorithm
5. Recursión por la izquierda

Gestión de errores



GESTIÓN DE ERRORES

- El propósito del compilador es
 - Para detectar programas no válidos
 - Traducir los válidos
- Muchos tipos de posibles errores (por ejemplo, en C)

Tipo	Ejemplo	Detectado por
Léxico	\$	Lexer
Sintáctico	x * %	Parser
Semántico	int x; y = x(3);	Type Checker
Exactitud	tu prox. proyecto de compi	Usuario



GESTIÓN DE ERRORES

- El manejador de errores debe:
 - Reportar los errores de manera precisa y clara.
 - Recuperarse rápidamente de un error.
 - No ralentizar la compilación de código válido.
- Un buen manejo de errores no es fácil de lograr.



GESTIÓN DE ERRORES

- Enfoques de simples a complejos:
 - Modo pánico
 - Producción de errores
 - Corrección automática local o global
- No todos son compatibles con todos los generadores de analizadores.



GESTIÓN DE ERRORES

Modo pánico

El modo de pánico es el método más simple y popular

Cuando se detecta un error:

- Descartar tokens hasta que se encuentre uno con un rol claro
- Continuar desde allí

Buscando tokens de sincronización

- Por lo general, los terminadores de declaraciones o expresiones



GESTIÓN DE ERRORES

Modo pánico

Considere la expresión errónea

(1 + +2) + 3

Recuperación en modo pánico: Saltar al siguiente entero y luego continuar

Bison/Cup: use el error de terminal especial para describir cuanto saltar en la entrada

$E \rightarrow \text{int} \mid E + E \mid (E) \mid \text{error} \mid \text{int} \mid \text{error}$



GESTIÓN DE ERRORES

Modo pánico

Ejemplo, entrada:

```
int 123a = 5;
```

```
float x = 3.14.15;
```

```
string name = "Hello World;
```



GESTIÓN DE ERRORES

Modo pánico

Lexer, macros:

letter = [a-zA-Z]

digit = [0-9]

oper = [+*/]

whitespace = [\t\n\r]

separator = [;()""]



GESTIÓN DE ERRORES

Modo pánico

Lexer, expresiones regulares:

id = letter (digit | letter)*

num = digit+ (.digit+)?

no_recognized =[^a-zA-Z0-9+/-*/ \t\n\r;()"]

malformed = digit+ letter+ digit* | digit* .[^digit]+



GESTIÓN DE ERRORES

Modo pánico

Parser, producciones SIN manejo de errores:

S -> ASSIGN;

ASSIGN -> id = EXPR

EXPR -> EXPR + TERM | EXPR - TERM | TERM

TERM -> TERM * FACTOR | TERM / FACTOR | FACTOR

FACTOR -> (EXPR) | num | id



GESTIÓN DE ERRORES

Modo pánico

Parser, producciones **CON** manejo de errores:

S -> ASSIGN; | *error ';'*

ASSIGN -> id = EXPR | *error '=' EXPR*

EXPR -> EXPR + TERM | EXPR - TERM | TERM | *error ('+' | '-')*

TERM -> TERM * FACTOR | TERM / FACTOR | FACTOR | *error ('*' | '/')*

FACTOR -> (EXPR) | num | id | *error ('(' | ')' | num | id)*



GESTIÓN DE ERRORES

Producciones de error

Especificar errores comunes conocidos en la gramática

Ejemplo:

- Escribe 5 x en lugar de 5 * x
- Añadir la producción $E \rightarrow E \ E$

Desventaja:

- Complica la gramática



GESTIÓN DE ERRORES

Producciones de error

Otros ejemplos:

expresiones donde falta un paréntesis de cierre, como en **5 * (3 + 2**

Añadir la producción $E \rightarrow (E$

capturar errores donde un operador es repetido innecesariamente, como en **5 ++ 3**

Añadir la producción $E \rightarrow E ++ E$



GESTIÓN DE ERRORES

Corrección automática local o global

- Idea: encontrar un programa “cercano” correcto
 - Pruebe las inserciones y eliminaciones de tokens
 - Búsqueda exhaustiva
- Desventajas:
 - Difícil de implementar
 - Ralentiza el análisis de los programas correctos
 - “Cercano” no es necesariamente el programa “previsto”



GESTIÓN DE ERRORES

- Pasado
 - Ciclo de recompilación lento (incluso una vez al día)
 - Encuentra tantos errores en un ciclo como sea posible
- Presente
 - Ciclo de recopilación rápida
 - Los usuarios tienden a corregir un error/ciclo
 - La recuperación de errores complejos es menos convincente

AST

(Abstract Syntax Trees)



AST

Un parser rastrea la derivación de una secuencia de tokens.

Pero el resto del compilador necesita una representación estructural del programa.

Árboles de sintaxis abstracta (Abstract Syntax Trees)

- Como los parse trees vistos hasta ahora pero ignorando ciertos detalles.
- Abreviado como AST



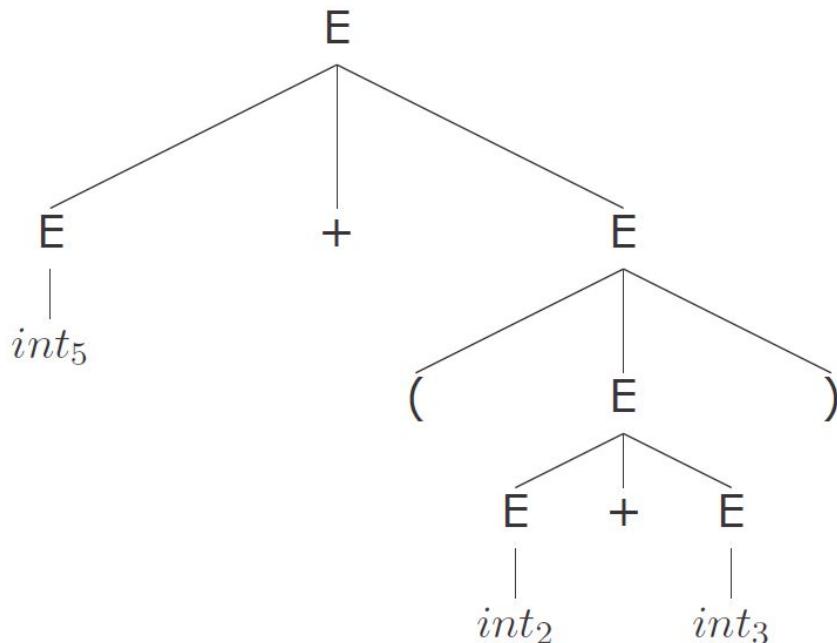
AST

- Considere la gramática
 - $E \rightarrow \text{int} \mid (E) \mid E + E$
- Y la cadena
 - $5 + (2 + 3)$
- Después del análisis léxico (una lista de tokens)
 - $\text{int5} + (\text{int2} + \text{int3})$



Ejemplo de Parse Tree

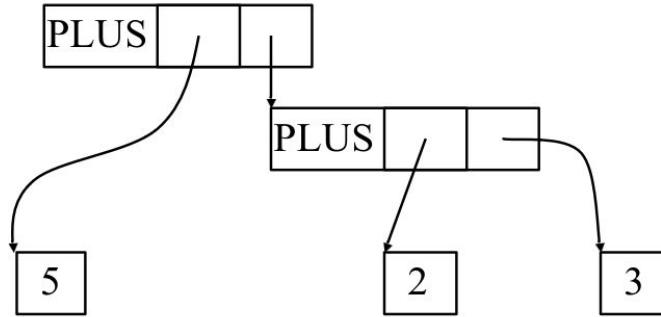
Durante el análisis construimos un árbol de análisis



- Un árbol de análisis
 - Rastrea el funcionamiento del parser.
 - Captura la estructura anidada
 - Mucha información (paréntesis, sucesores simples)



Ejemplo de Abstract Syntax Tree



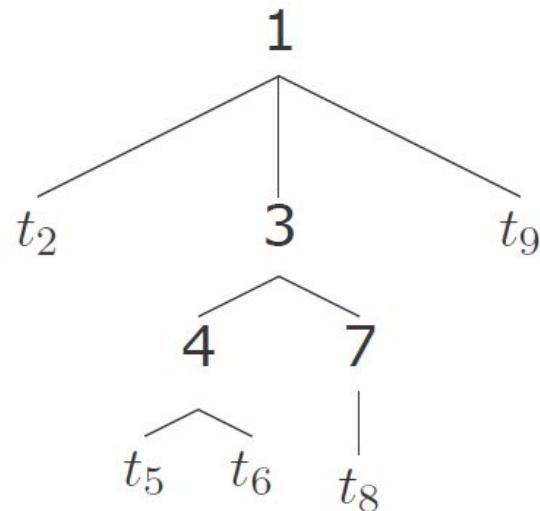
- También captura la estructura de anidamiento
- Pero se abstrae de la sintaxis concreta
 - Más compacto y fácil de usar
- Es una estructura de datos importante en un compilador

Recursive Descent Parsing



RECURSIVE DESCENT PARSING

- El árbol de análisis se construye
 - Desde la parte superior
 - De izquierda a derecha
- Los terminales se ven en orden de aparición en el flujo de tokens: **t2 t5 t6 t8 t9**





RECURSIVE DESCENT PARSING

- Considere la gramática
 - $E \rightarrow T \mid T + E$
 - $T \rightarrow int \mid int * T \mid (E)$
- El flujo de tokens es: (*int5*)
- Comience con el nivel superior no terminal E
- Pruebe las reglas para E en orden

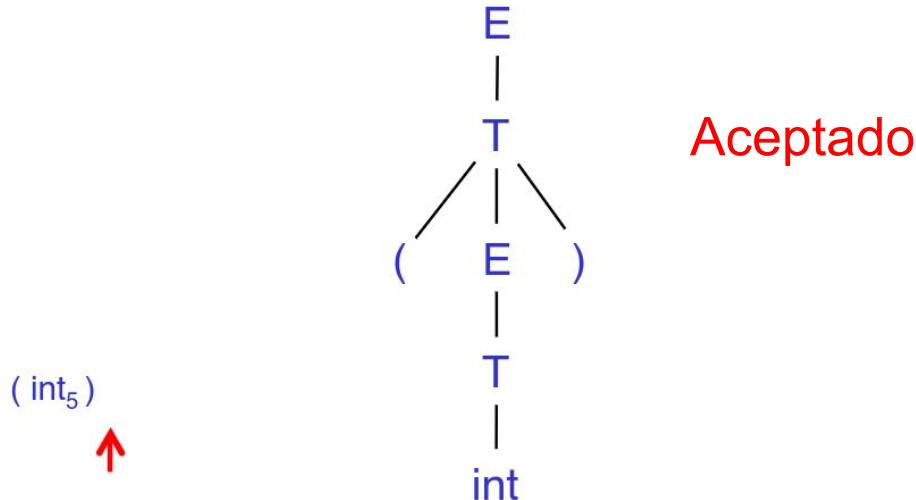


RECURSIVE DESCENT PARSING

$$E \rightarrow T \mid T + E$$
$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$
$$E$$
$$(\text{int}_5)$$




RECURSIVE DESCENT PARSING

$$E \rightarrow T \mid T + E$$
$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$


Recursive Descent Algorithm



Recursive Descent Algorithm (RDA)

Consideraciones a tomar

- TOKEN: Representa un símbolo genérico que puede ser cualquier tipo de token -e.g. INT, OPEN, CLOSE, PLUS, TIMES -
- NEXT: Es un puntero o cursor que apunta al próximo token en la secuencia de entrada que se está analizando.



Recursive Descent Algorithm (RDA)

RDA es un algoritmo que define **funciones booleanas** que verifiquen coincidencia de:

- 1) Un terminal de token dado

```
bool term(TOKEN tok) return *next++ == tok;
```

verifica si el token al que apunta NEXT coincide con el token esperado (tok)



Recursive Descent Algorithm (RDA)

- 2) La enésima producción de S

```
bool Sn() ...
```

- 3) Comprueba todas las producciones de S:

```
bool S() ...
```



Recursive Descent Algorithm (RDA)

Considerando la gramática:

$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow \text{int}$

$T \rightarrow \text{int} * T$

$T \rightarrow (E)$



Recursive Descent Algorithm (RDA)

Para la producción $E \rightarrow T$

```
bool E1() { return T(); }
```

Para la producción $E \rightarrow T + E$

```
bool E2() { return T() && term(PLUS) && E(); }
```



Recursive Descent Algorithm (RDA)

Para todas las producciones de E (con backtracking)

```
bool E() {  
  
TOKEN *save = next; //guarda el valor actual del puntero  
  
return (next = save, E1()) || (next = save, E2()); }
```



Recursive Descent Algorithm (RDA)

Funciones para un no terminal T

```
bool T1() { return term(INT); }

bool T2() { return term(INT) && term(TIMES) && T(); }

bool T3() { return term(OPEN) && E() && term(CLOSE); }

bool T() {
    TOKEN *save = next;

    return (next = save, T1()) || (next = save, T2()) ||
           (next = save, T3());
}
```



Recursive Descent Algorithm (RDA)

Para iniciar el analizador

- Inicializar al lado del punto del primer token
 - Invocar E()
- Fácil de implementar a mano

Recursive Descent Algorithm (RDA)



Recursive Descent Algorithm (RDA)

$$\begin{aligned} E &\rightarrow E' | E' + id \\ E' &\rightarrow -E' | id | (E) \end{aligned}$$

- Línea 3
- Línea 5
- Línea 6
- Línea 12

```
1 bool term(TOKEN tok) { return *next++ == tok; }

2 bool E1() { return E'(); }

3 bool E2() { return E'() && term(PLUS) && term(ID); }

4 bool E() {

5     TOKEN *save = next;

6     return (next = save, E1()) && (next = save, E2());

7 }

8 bool E'1() { return term(MINUS) && E'(); }

9 bool E'2() { return term(ID); }

10 bool E'3() { return term(OPEN) && E() && term(CLOSE); }

11 bool E'() {

12     TOKEN *next = save; return (next = save, T1()) || (next = save, T2()) || (next = save, T3());

13

14

15 }
```

Recursión por la izquierda



Recursión por la izquierda

- Considere una producción $S \rightarrow S\alpha$

```
bool S1() { return S() && term(a); }
```

```
bool S() { return S1(); }
```

- $S()$ entra en un ciclo infinito
- Una gramática recursiva por la izquierda tiene una S no terminal $S \rightarrow^+ S\alpha$ para alguna α
- El descenso recursivo **no funciona** en tales casos



Recursión por la izquierda

- Considere la gramática recursiva por la izquierda
- $$S \rightarrow S\alpha | \beta$$
- S genera todas las cadenas que comienzan con β y siguen cualquier número de α
 - Se puede reescribir usando la recursividad derecha

$$S \rightarrow \beta S O$$

$$SO \rightarrow \alpha SO | \epsilon$$



Recursión por la izquierda

En general:

$$S \rightarrow S\alpha_1 | \dots | S\alpha_n | \beta_1 | \dots | \beta_m$$

Todas las cadenas derivadas de S comienzan con uno de β_1, \dots, β_m y continúan con varias instancias de $\alpha_1, \dots, \alpha_n$

Reescribir como

$$S \rightarrow \beta_1 S' | \dots | \beta_m S'$$

$$S' \rightarrow \alpha_1 S' | \dots | \alpha_n S' | \epsilon$$



Recursión por la izquierda

Considere la siguiente gramática:

E → **E** + **T**

| **T**

T → **INT**

lo que puede llevar a un bucle infinito al intentar analizar una expresión como 1 + 2 + 3



Recursión por la izquierda

La gramática:

$$S \rightarrow A \alpha \mid \delta$$

$$A \rightarrow S \beta$$

También es recursiva por la izquierda, dado que si reemplazamos la producción A en S tendremos:

$$S \rightarrow^+ S \beta \alpha$$



Recursión por la izquierda

Acerca del Descenso Recursivo

- Estrategia de análisis simple y general
 - La recursividad a la izquierda debe eliminarse primero
 - ... pero eso se puede hacer automáticamente
- Históricamente impopular debido al **backtracking**.
 - Se pensaba que era demasiado ineficiente.
 - En la práctica, con algunos ajustes, es rápida y simple en máquinas modernas.
 - El **backtracking** puede ser controlado restringiendo la gramática.



Recursión por la izquierda



PARSERS PREDICTIVOS

— Ing. Max Cerna —

Agenda

1. Predictive Top-Down Parsers
2. First
3. Follow
4. Tabla LL(1)

Predictive Top-Down Parsers

PREDICTIVE TOP-DOWN PARSERS

- Analizadores predictivos de arriba hacia abajo
- Similares a RDP pero el analizador puede "predecir" qué producción usar
 - Mirando los siguientes tokens
 - Sin retroceso (backtrack)
- Los analizadores predictivos aceptan gramáticas LL(k)
 - L significa escaneo de entrada "de izquierda a derecha"
 - L significa "derivación (más a la/por) izquierda"
 - k significa "predecir basado en k tokens de anticipación"
 - En la práctica, se utiliza LL(1)

PREDICTIVE TOP-DOWN PARSERS

- Por ejemplo, recordemos la gramática

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

- Difícil de predecir porque
 - Para T dos producciones comienzan con int
 - Para E no está claro cómo predecir
- Necesitamos factorizar la gramática por/(a la) izquierda

Ejemplo de factorización a la izquierda

Factorizar los prefijos comunes de las producciones

$E \rightarrow T \ x$ // factor común T, lo de la derecha genera nueva producción

$x \rightarrow + E \mid \epsilon$ // nueva producción

$T \rightarrow int \ y \mid (E)$ // factor int, derecha genera nueva producción

$y \rightarrow * T \mid \epsilon$ // nueva producción

PREDICTIVE TOP-DOWN PARSERS

Elija la alternativa que factoriza correctamente la gramática dada

```
EXPR → if BOOL then { EXPR }
| if BOOL then { EXPR } else { EXPR }
| ...
BOOL → true | false
```

1.

```
EXPR → if true then { EXPR }
| if false then { EXPR }
| if true then { EXPR } else { EXPR }
| if false then { EXPR } else { EXPR }
| ...
```

2.

```
EXPR → if BOOL EXPR'
| ...
EXPR' → then { EXPR }
| then { EXPR } else { EXPR }
BOOL → true | false
```

3.

```
EXPR → EXPR' | EXPR' else { EXPR }
EXPR' → if BOOL then { EXPR }
| ...
BOOL → true | false
```

4.

```
EXPR → if BOOL then { EXPR } EXPR'
| ...
EXPR' → else { EXPR } | ε
BOOL → true | false
```

PREDICTIVE TOP-DOWN PARSERS

Gramática factorizada

$$E \rightarrow TX$$

$$T \rightarrow (E) | int Y$$

$$X \rightarrow +E | \varepsilon$$

$$Y \rightarrow *T | \varepsilon$$

Tabla LL(1)

siguiente token de entrada

proxima produccion a utilizar

no terminal

	int	*	+	()	\$
E	TX			TX		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

PREDICTIVE TOP-DOWN PARSERS

Considere la entrada $[E, \text{int}]$

- “Cuando el no terminal actual es E y la siguiente entrada es int , usar la producción $E \rightarrow TX$ ”
- Esto puede generar un int en la primera posición

	int	*	+	()	\$
E	TX			TX		
X			+ E		ϵ	ϵ
T	int Y			(E)		
Y		* T	ϵ		ϵ	ϵ

PREDICTIVE TOP-DOWN PARSERS

Considere la entrada $[Y, +]$

- “Cuando el no terminal actual es Y y la siguiente entrada es $+$, eliminar Y ”
- Y puede ir seguido de $+$ solo si $Y \rightarrow \epsilon$

	int	*	+	()	\$
E	TX			TX		
X			+ E		ϵ	ϵ
T	int Y			(E)		
Y		* T	ϵ		ϵ	ϵ



PREDICTIVE TOP-DOWN PARSERS

Considere la entrada $[E, *]$

- “No hay forma de derivar una cadena que comience con $*$ desde el no terminal E ”

	int	*	+	()	\$
E	TX			TX		
X			+ E		ϵ	ϵ
T	int Y			(E)		
Y		* T	ϵ		ϵ	ϵ

PREDICTIVE TOP-DOWN PARSERS

- Método similar al descenso recursivo, excepto
 - Para el S no terminal más a la izquierda
 - Miramos el siguiente token de entrada a
 - Y elige la producción que se muestra en $[S, a]$
- Una pila registra la frontera del árbol de análisis
 - No terminales que aún no se han ampliado
 - Terminales que aún tienen que coincidir con la entrada
 - Parte superior de la pila = terminal pendiente más a la izquierda o no terminal
- Rechazar al llegar al estado de error
- Aceptar al final de la entrada y pila vacía

LL(1) Parsing Algorithm

initialize stack = $\langle S \ $ \rangle$ and next

repeat

 case stack of

$\langle X, \text{rest} \rangle$: if $T[X, *next] = Y_1 \dots Y_n$

 then stack $\leftarrow \langle Y_1 \dots Y_n, \text{rest} \rangle$;

 else error ();

$\langle t, \text{rest} \rangle$: if $t == *next ++$

 then stack $\leftarrow \langle \text{rest} \rangle$;

 else error ();

until stack == $\langle \rangle$

LL(1) Parsing Algorithm

```
initialize stack = <S $> and next  
repeat
```

```
    case stack of
```

```
        <X, rest> : if T[X,*next] = Y1...Yn  
                      then stack ← <Y1...Yn, rest>;  
                      else error ();
```

```
        <t, rest> : if t == *next ++
```

```
                      then stack ← <rest>;  
                      else error ();
```

Para la terminal t en la parte superior de la pila, verifique que t coincida con el siguiente token de entrada.

```
until stack == <>
```

marca el fondo de la pila

Para X no terminal en la parte superior de la pila, búsqueda de producción

Pop X , push la producción a la pila. Ten en cuenta que el símbolo más a la izquierda de la producción está en la parte superior de la pila.

Ejemplo LL(1) Parsing

	int	*	+	()	\$
E	TX			TX		
X			+ E		ϵ	ϵ
T	int Y			(E)		
Y		* T	ϵ		ϵ	ϵ

Stack	Entrada	Acción
E \$	int * int \$	TX
TX \$	int * int \$	int Y
int Y X \$	int * int \$	terminal
Y X \$	* int \$	* T
* TX \$	* int \$	terminal
TX \$	int \$	int Y
int Y X \$	int \$	terminal
Y X \$	\$	ϵ
X \$	\$	ϵ
\$	\$	ACCEPT

PREDICTIVE TOP-DOWN PARSERS

Considere la siguiente tabla y gramática. ¿Cuál es el siguiente estado si el stack actualmente contiene $E' \$$ y la entrada contiene

else { if false then { false } } \$?

	if	then	else	{	}	true	false	\$
E	if B then { E } E'			ϵ	B	B	ϵ	
E'			else { E }	ϵ			ϵ	
B						true	false	

$E \rightarrow \text{if } B \text{ then } \{E\} \ E' \mid B \mid \epsilon$

$E' \rightarrow \text{else } \{E\} \mid \epsilon$

$B \rightarrow \text{true} \mid \text{false}$

La Intuición para la construcción de tablas de análisis

- Considere el no terminal A , la producción $A \rightarrow \alpha$ y el token t
- Agregar $T[A, t] = \alpha$ en dos casos
 1. Si $A \rightarrow \alpha \rightarrow^* t\beta$
 - α puede derivar t en la primera posición
 - Decimos que $t \in First(\alpha)$
 2. Si $A \rightarrow \alpha \rightarrow^* \epsilon$ y $S \rightarrow^* \gamma A t \delta$
 - Útil si la pila tiene A , la entrada es t y A no puede derivar t
 - En este caso, la única opción es deshacerse de A (derivando ϵ)
 - Lo anterior solo puede funcionar si t puede seguir a A en al menos una derivación
 - Decimos $t \in Follow(A)$

First

First

Definición:

$$\text{First}(X) = \{t \mid X \rightarrow t\alpha\} \cup \{\varepsilon \mid X \rightarrow \varepsilon\}$$

Algoritmo:

- 1) $\text{First}(t) = \{t\}$
- 2) $\varepsilon \in \text{First}(X)$
 - a) si $X \rightarrow \varepsilon$
 - b) si $X \rightarrow A_1 \dots A_n$ y $\varepsilon \in \text{First}(A_i)$ para todo $1 \leq i \leq n$
- 3) $\text{First}(\alpha) \subseteq \text{First}(X)$
 - a) si $X \rightarrow \alpha$
 - b) si $X \rightarrow A_1 \dots A_n \alpha$ y $\varepsilon \in \text{First}(A_i)$ para todo $1 \leq i \leq n$

First

Trabajemos con la gramática factorizada

$$E \rightarrow T \ X$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$X \rightarrow + E \mid \epsilon$$

$$Y \rightarrow * T \mid \epsilon$$

Ejercicio 1 - First

$S \rightarrow A \ B \ C$

$A \rightarrow x \ A$

| ϵ

$B \rightarrow y \ B$

| ϵ

$C \rightarrow z$

Ejercicio 2 - First

$S \rightarrow a A$

| $b B$

$A \rightarrow c A$

| d

$B \rightarrow \epsilon$

Follow

Follow

Definición:

$$\text{Follow}(X) = \{t \mid S \xrightarrow{*} \beta X t \delta\}$$

Razonamiento

- Si $X \rightarrow AB$ entonces $\text{First}(B) \subseteq \text{Follow}(A)$ y $\text{Follow}(X) \subseteq \text{Follow}(B)$
 - También si $B \xrightarrow{*} \epsilon$ entonces $\text{Follow}(X) \subseteq \text{Follow}(A)$
- Si S es el símbolo inicial entonces $\$ \in \text{Follow}$

Follow

Algoritmo:

1. $\$ \in \text{Follow}(S)$
2. $\text{First}(\beta) - \{\varepsilon\} \subseteq \text{Follow}(X)$
 - a. Para cada producción $A \rightarrow \alpha X \beta$
3. $\text{Follow}(A) \subseteq \text{Follow}(X)$
 - a. Para cada producción $A \rightarrow \alpha X \beta$ donde $\varepsilon \in \text{First}(\beta)$

Ejercicio 1 - Follow

$S \rightarrow X Y$

$X \rightarrow a X \mid \epsilon$

$Y \rightarrow b \mid \epsilon$

Ejercicio 2 - Follow

$S \rightarrow A \ B$

$A \rightarrow a \ A \mid \epsilon$

$B \rightarrow b \ B \mid c$

Tabla LL(1)

Tabla LL(1)

Construir una tabla de análisis T para CFG

Para cada producción $A \rightarrow \alpha$ en G debemos:

- Para cada terminal $t \in \text{First}(\alpha)$ colocamos $T[A, t] = \alpha$
- Si $\epsilon \in \text{First}(\alpha)$, para cada $t \in \text{Follow}(A)$ colocamos $T[A, t] = \alpha$
- Si $\epsilon \in \text{First}(\alpha)$ y $\$ \in \text{Follow}(A)$ colocamos $T[A, \$] = \alpha$

Tabla LL(1)

Trabajemos con la gramática factorizada

$$E \rightarrow T X$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$X \rightarrow + E \mid \epsilon$$

$$Y \rightarrow * T \mid \epsilon$$

Tabla LL(1)

No Terminal	First	Follow
E	{int, (}	{ \$,) }
X	{ +, ε }	{ \$,) }
T	{ int, (}	{ +, \$,) }
Y	{ *, ε }	{ +, \$,) }

Ejemplo LL(1) Parsing

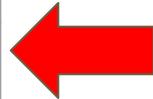
$$E \rightarrow T \ X$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$X \rightarrow + E \mid \varepsilon$$

$$Y \rightarrow * T \mid \varepsilon$$

	int	*	+	()	\$
E	T X			T X		
X						
T						
Y						



No Terminal	First	Follow
E	{int, (}	{ \$,) }
X	{ +, ε }	{ \$,) }
T	{ int, (}	{ +, \$,) }
Y	{ *, ε }	{ +, \$,) }

Ejemplo LL(1) Parsing

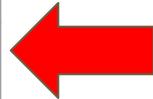
$$E \rightarrow T \ X$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$X \rightarrow + E \mid \varepsilon$$

$$Y \rightarrow * T \mid \varepsilon$$

	int	*	+	()	\$
E	T X			T X		
X			+ E			
T						
Y						



No Terminal	First	Follow
E	{int, (}	{ \$,) }
X	{ +, ε }	{ \$,) }
T	{ int, (}	{ +, \$,) }
Y	{ *, ε }	{ +, \$,) }

Ejemplo LL(1) Parsing

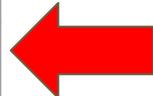
$$E \rightarrow T X$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$X \rightarrow + E \mid \epsilon$$

$$Y \rightarrow * T \mid \epsilon$$

	int	*	+	()	\$
E	T X			T X		
X			+ E			
T	int Y					
Y						



No Terminal	First	Follow
E	{int, (}	{ \$,) }
X	{ +, ε }	{ \$,) }
T	{ int, (}	{ +, \$,) }
Y	{ *, ε }	{ +, \$,) }

Ejemplo LL(1) Parsing

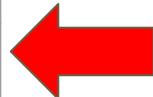
$$E \rightarrow T \ X$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$X \rightarrow + E \mid \varepsilon$$

$$Y \rightarrow * T \mid \varepsilon$$

	int	*	+	()	\$
E	T X			T X		
X			+ E			
T	int Y			(E)		
Y						



No Terminal	First	Follow
E	{int, (}	{ \$,) }
X	{ +, ε }	{ \$,) }
T	{ int, (}	{ +, \$,) }
Y	{ *, ε }	{ +, \$,) }

Ejemplo LL(1) Parsing

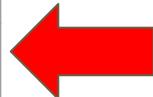
$$E \rightarrow T \ X$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$X \rightarrow + E \mid \varepsilon$$

$$Y \rightarrow * T \mid \varepsilon$$

	int	*	+	()	\$
E	T X			T X		
X			+ E			
T	int Y			(E)		
Y		* T				



No Terminal	First	Follow
E	{int, (}	{ \$,) }
X	{ +, ε }	{ \$,) }
T	{ int, (}	{ +, \$,) }
Y	{ *, ε }	{ +, \$,) }

Ejemplo LL(1) Parsing

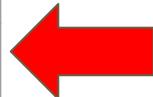
$$E \rightarrow T \ X$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$X \rightarrow + E \mid \epsilon$$

$$Y \rightarrow * T \mid \epsilon$$

	int	*	+	()	\$
E	T X			T X		
X			+ E		ϵ	ϵ
T	int Y			(E)		
Y		* T				



No Terminal	First	Follow
E	{int, (}	{ \$,) }
X	{ +, ϵ }	{ \$,) }
T	{ int, (}	{ +, \$,) }
Y	{ *, ϵ }	{ +, \$,) }

Ejemplo LL(1) Parsing

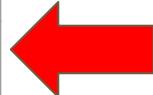
$$E \rightarrow T \ X$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$X \rightarrow + E \mid \epsilon$$

$$Y \rightarrow * T \mid \epsilon$$

	int	*	+	()	\$
E	T X			T X		
X			+ E		ϵ	ϵ
T	int Y			(E)		
Y		* T	ϵ		ϵ	ϵ



No Terminal	First	Follow
E	{int, (}	{ \$,) }
X	{ +, ϵ }	{ \$,) }
T	{ int, (}	{ +, \$,) }
Y	{ *, ϵ }	{ +, \$,) }

Ejercicio 1 - Tabla LL(1)

$S \rightarrow X Y$

$X \rightarrow a X \mid \epsilon$

$Y \rightarrow b \mid \epsilon$

No Terminal	First	Follow
S	{ a, b, ϵ }	{ \$, b }
X	{ a, ϵ }	{ b }
Y	{ b, ϵ }	{ \$, b }

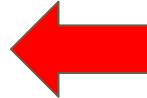
Ejercicio 1 - Tabla LL(1)

$$S \rightarrow X Y$$

$$X \rightarrow a X \mid \epsilon$$

$$Y \rightarrow b \mid \epsilon$$

	a	b	\$
S	XY	XY	XY
X	a X	ϵ	
Y		b/ ϵ	ϵ



No Terminal	First	Follow
S	{ a, b, ϵ }	{ \$, b }
X	{ a, ϵ }	{ b }
Y	{ b, ϵ }	{ \$, b }

Ejercicio 1 - Tabla LL(1)

$$S \rightarrow A \ B$$

$$A \rightarrow a \ A \mid \epsilon$$

$$B \rightarrow b \ B \mid c$$

No Terminal	First	Follow
S	{ a, b, c }	{ \$ }
A	{ a, ε }	{ b, c }
B	{ b, c }	{ \$ }

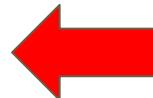
Ejercicio 1 - Tabla LL(1)

$$S \rightarrow A \ B$$

$$A \rightarrow a \ A \mid \epsilon$$

$$B \rightarrow b \ B \mid c$$

	a	b	c	\$
S	AB			AB
A	aA	ϵ	ϵ	
B		b B	c	



No Terminal	First	Follow
S	{ a, b, c }	{ \$ }
A	{ a, ϵ }	{ b, c }
B	{ b, c }	{ \$ }

Tabla LL(1)

- Si una entrada es definida múltiples veces entonces G no es LL(1)
 - No está factorizada por la izquierda
 - Tiene recursividad por la izquierda
 - Es ambigua
- La mayoría de lenguajes de programación no son LL(1)

Ejemplos y Ejercicios para CFG

Ing. Max Cerna





Recursividad por la izquierda

Si tenemos producciones de la forma:

$$A \rightarrow Aa \mid b$$

Estas dos producciones pueden ser sustituidas por:

$$A \rightarrow bA'$$

$$A' \rightarrow aA' \mid \epsilon$$



Ejemplo

Remover recursividad por la izquierda de la siguiente gramatica:

$$S \rightarrow Ab \mid b$$

$$A \rightarrow Ac \mid Sd \mid x$$

recursividad indirecta de **S**, vamos a sustituir la producción **S** en **A** → **Sd** usando las producciones de **S** → **Ab** | **b**

Ahora las producciones de A seran

$$A \rightarrow Ac \mid Abd \mid bd \mid x$$



Ejemplo

Introducir una nueva variable para eliminar la recursividad por la izquierda en A, que llamaremos A'.

Dividimos las producciones de A

- Las que comienzan con A (recursivas): $A \rightarrow Ac \mid Abd$
- Las que no comienzan con A (no recursivas): $A \rightarrow bd \mid x$



Ejemplo

Las producciones de A se convierten en las partes no recursivas seguidas por A':

$$A \rightarrow bdA' \mid xA'$$

Las producciones de A' manejarán las partes recursivas:

$$A' \rightarrow cA' \mid bdA' \mid \epsilon$$



Ejemplo

Gramática final

S → **Ab** | **b**

A → **bdA'** | **xA'**

A' → **cA'** | **bdA'** | **ε**



Ejercicio

A → Aa | A1 | A2 | b

B → Bb | A

C → Cc | D

D → d | e



Ejercicio

¿Cuántas cadenas pertenecen al lenguaje descrito por la siguiente gramática?

$$S \rightarrow AB$$

$$A \rightarrow 0 \mid \epsilon$$

$$B \rightarrow 1B \mid 1 \mid \epsilon$$



Ejercicio

¿Cuántas cadenas pertenecen al lenguaje descrito por la siguiente gramática?

$$S \rightarrow AB$$

$$A \rightarrow 0 \mid 1 \mid \epsilon$$

$$B \rightarrow 1 \mid 2 \mid \epsilon$$



Ejercicio

¿Las gramáticas son equivalentes?

Gramática 1: $S \rightarrow aSb \mid \epsilon$

Gramática 2: $T \rightarrow aTb \mid ab \mid \epsilon$



Ejercicio

Dada la gramática recursiva por la izquierda:

$$S \rightarrow Sa \mid b$$

Cuales gramáticas quitan la recursividad por la izquierda y son equivalentes:

a) $S \rightarrow bA$
 $A \rightarrow aA \mid \epsilon$

b) $S \rightarrow b \mid bS$

c) $S \rightarrow bA$
 $A \rightarrow a \mid aa$

d) $S \rightarrow A \mid b$
 $A \rightarrow aA \mid a$

PARSERS BOTTOM-UP

Ing. Max Cerna





Agenda

1. Parsers bottom-up
2. Shift-reduce
3. Handles
4. Trabajando con Handles
5. Reconociendo prefixes Viables
6. Parsing LR(0) y SLR

Parsers bottom-up



PARSERS BOTTOM-UP

- El análisis de abajo hacia arriba es más general que análisis de arriba hacia abajo (determinista)
- Igual de eficiente
- Se basa en ideas en el análisis de arriba hacia abajo
- De abajo hacia arriba es el método preferido por distintas herramientas (CUP, Bison)



PARSERS BOTTOM-UP

- Los analizadores de abajo hacia arriba no necesitan factorizar a la izquierda.
- Vuelve a la gramática “natural”

$E \rightarrow T + E$

| T

$T \rightarrow int$

| $int * T$

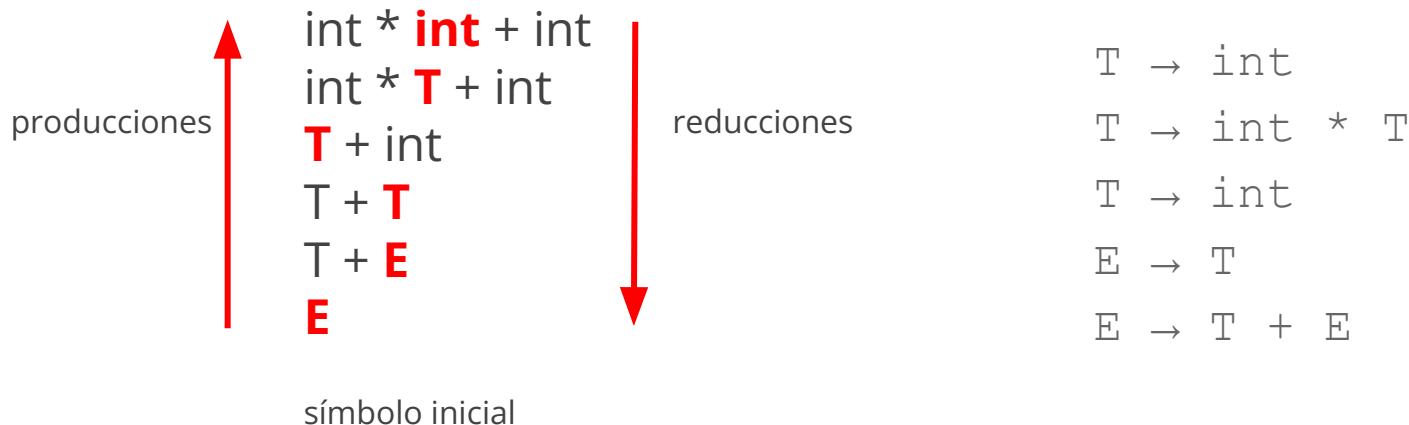
| (E)

- Consideremos la cadena $int * int + int$



PARSERS BOTTOM-UP

- El análisis de abajo hacia arriba reduce una cadena al símbolo de inicio por inversión de producciones.





PARSERS BOTTOM-UP

Si las reducciones se leen al contrario, se puede trazar una derivación más a la derecha.

Regla #1

Un parser bottom-up traza una derivación más a la derecha/por la derecha

int * int + int

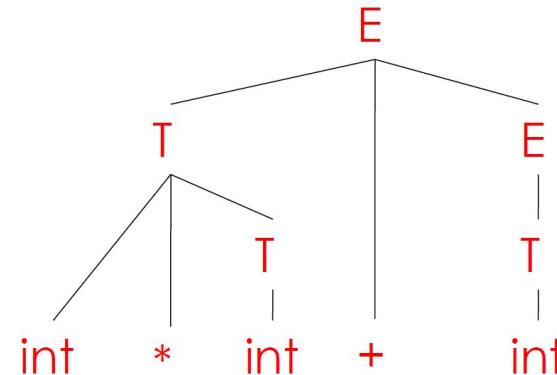
int * T + int

T + int

T + T

T + E

E





PARSERS BOTTOM-UP

→ int * int + int

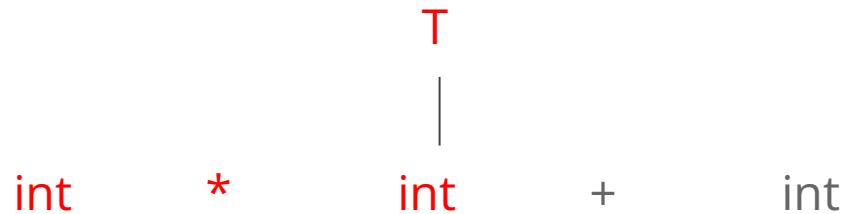
→ int * int + int



PARSERS BOTTOM-UP

int * int + int

int * T + int



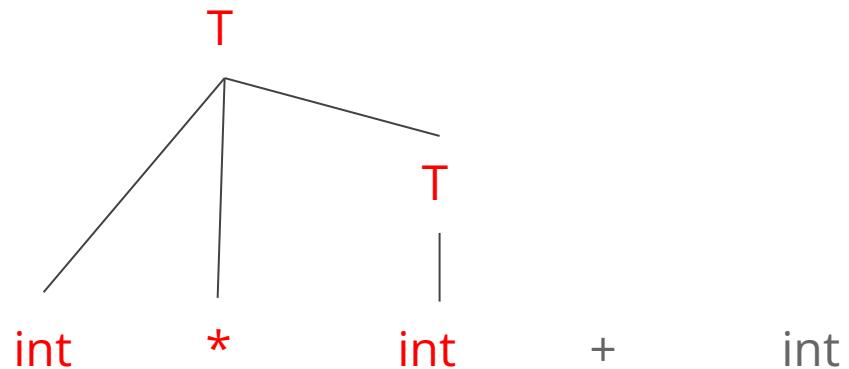


PARSERS BOTTOM-UP

int * int + int

int * T + int

T + int





PARSERS BOTTOM-UP

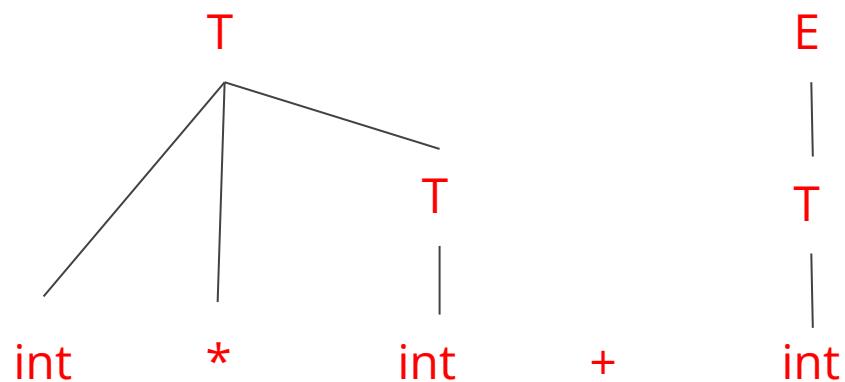
int * int + int

int * T + int

T + int

T + T

T + E





PARSERS BOTTOM-UP

int * int + int

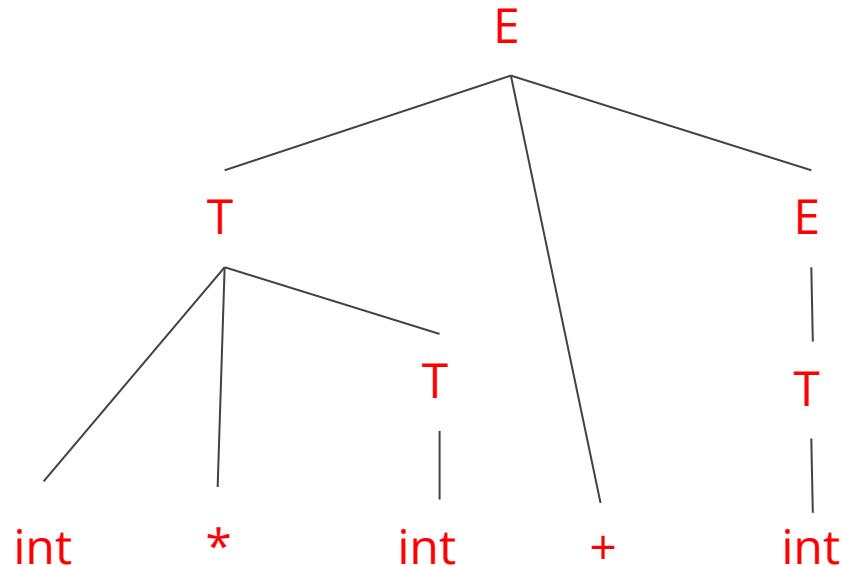
int * T + int

T + int

T + T

T + E

E





PARSERS BOTTOM-UP

Considere la siguiente gramática y genere las reducciones para la cadena
- (id+id)+id

$$E \rightarrow E' \mid E' + E$$

$$E' \rightarrow -E' \mid id \mid (E)$$

Shift-Reduce



SHIFT-REDUCE

La regla #1 tiene una consecuencia interesante:

- Sea $\alpha\beta\omega$ un paso dentro de un análisis de abajo hacia arriba
- Suponga que la próxima reducción es por $X \rightarrow \beta$
- Entonces ω es una secuencia de terminales

¿Por qué? Dado que $\alpha X \omega \rightarrow \alpha \beta \omega$ es un paso en el extremo derecho derivación



SHIFT-REDUCE

- Idea: dividir la cadena en dos subcadenas
- La subcadena derecha aún no ha sido examinada por el análisis (una cadena de terminales)
- La subcadena izquierda tiene terminales y no terminales
- El punto de división está marcado por un |
- Inicialmente, toda la entrada está sin examinar: |x₁x₂ . . . x_n



SHIFT-REDUCE

Definición

Los parsers de abajo hacia arriba solo cuentan con dos operaciones:
1- Shift y 2- Reduce

Shift

Mover | un lugar a la derecha
 $ABC|xyz \Rightarrow ABCx|yz$

Reduce

Aplicar una producción inversa en el extremo derecho de la cadena izquierda. Si $A \rightarrow xy$ es una producción, entonces
 $Cbxy|ijk \Rightarrow CbA|ijk$



SHIFT-REDUCE

int * int + int	shift
int * int + int	shift
int * int + int	shift
int * int + int	reduce $T \rightarrow \text{int}$
int * T + int	reduce $T \rightarrow \text{int} * T$
T + int	shift
T + int	shift
T + int	reduce $T \rightarrow \text{int}$
T + T	reduce $E \rightarrow T$
T + E	reduce $E \rightarrow T + E$
E	



SHIFT-REDUCE

|int * int + int

int * int + int



SHIFT-REDUCE

|int * int + int

int |* int + int

int * int + int
 ↑



SHIFT-REDUCE

|int * int + int

int |* int + int

int * |int + int

int * int + int
 ↑



SHIFT-REDUCE

|int * int + int

int |* int + int

int * |int + int

int * int |+ int

int * int + int





SHIFT-REDUCE

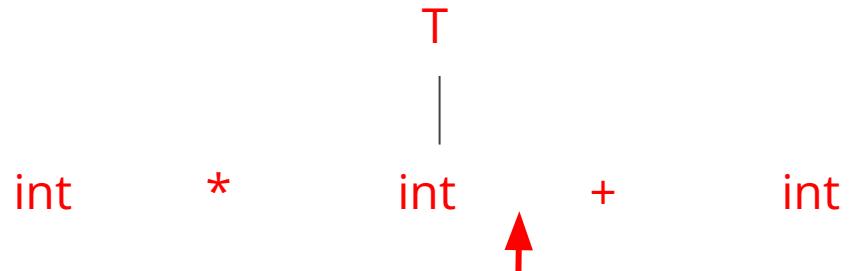
|int * int + int

int |* int + int

int * |int + int

int * int |+ int

int * T |+ int





SHIFT-REDUCE

|int * int + int

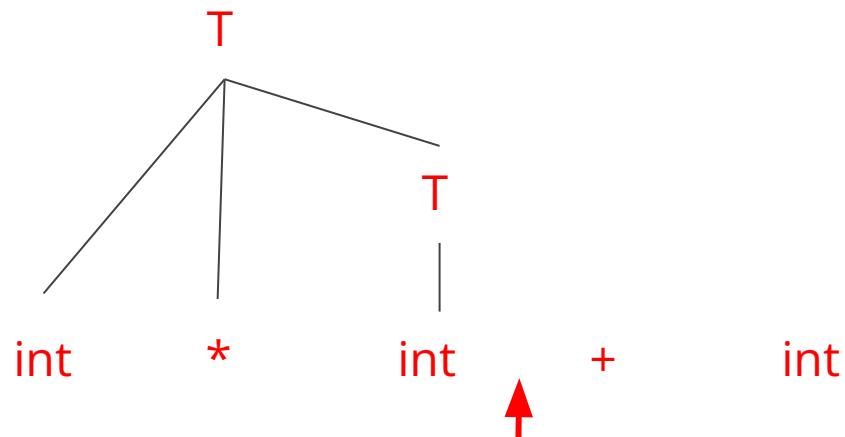
int |* int + int

int * |int + int

int * int |+ int

int * T |+ int

T |+ int





SHIFT-REDUCE

|int * int + int

int |* int + int

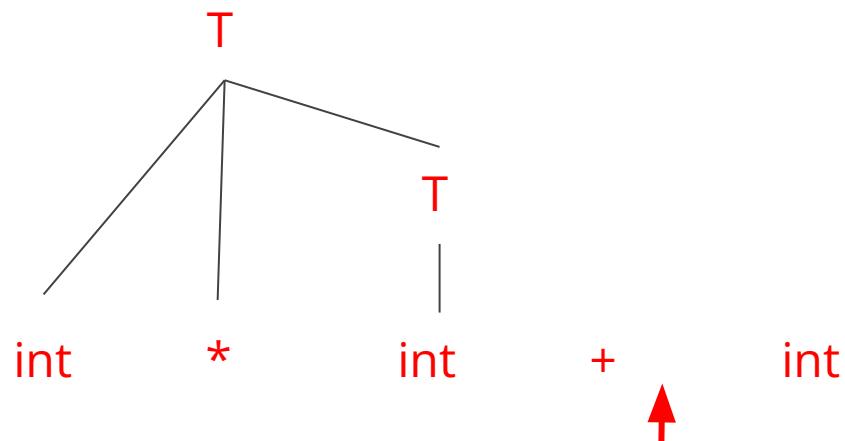
int * |int + int

int * int |+ int

int * T |+ int

T |+ int

T + |int





SHIFT-REDUCE

|int * int + int

int |* int + int

int * |int + int

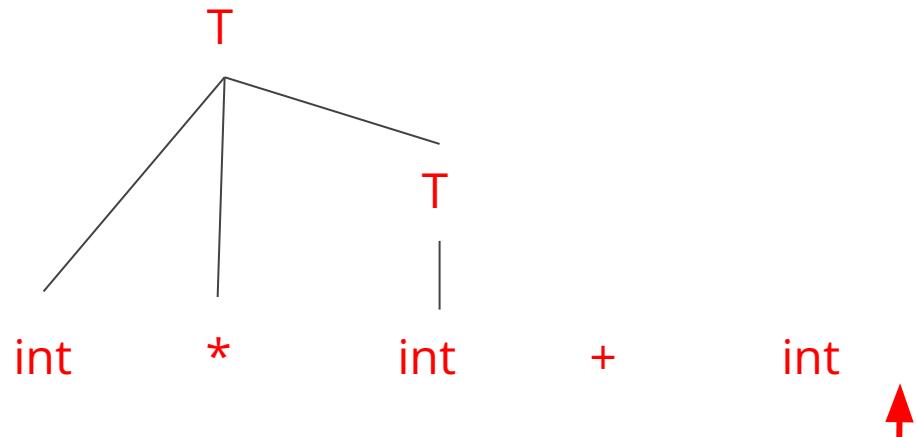
int * int |+ int

int * T |+ int

T |+ int

T + |int

T + int|





SHIFT-REDUCE

|int * int + int

int |* int + int

int * |int + int

int * int |+ int

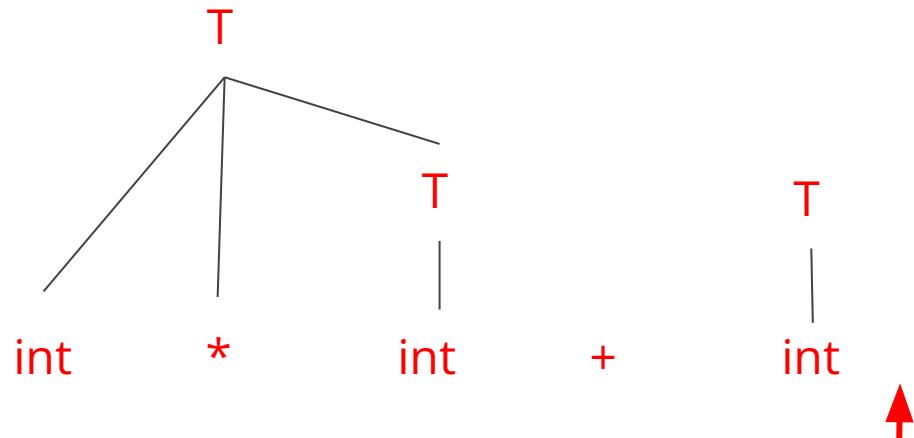
int * T |+ int

T |+ int

T + |int

T + int|

T + T|





SHIFT-REDUCE

|int * int + int

int |* int + int

int * |int + int

int * int |+ int

int * T |+ int

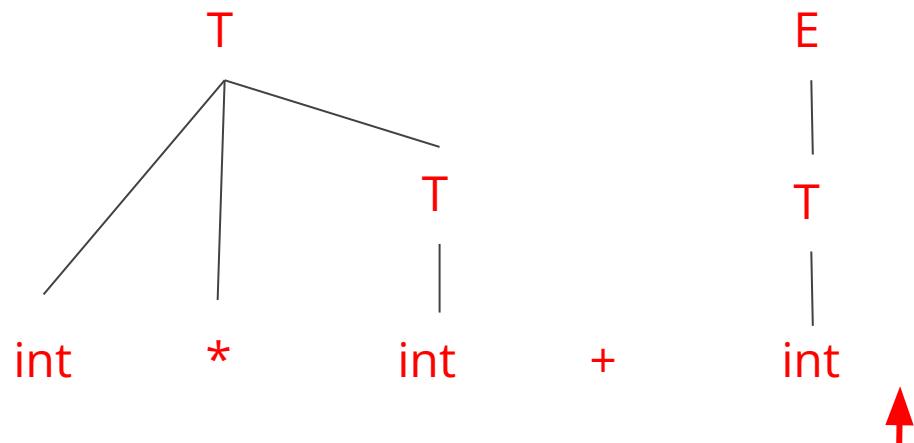
T |+ int

T + |int

T + int|

T + T|

T + E|





SHIFT-REDUCE

|int * int + int

int |* int + int

int * |int + int

int * int |+ int

int * T |+ int

T |+ int

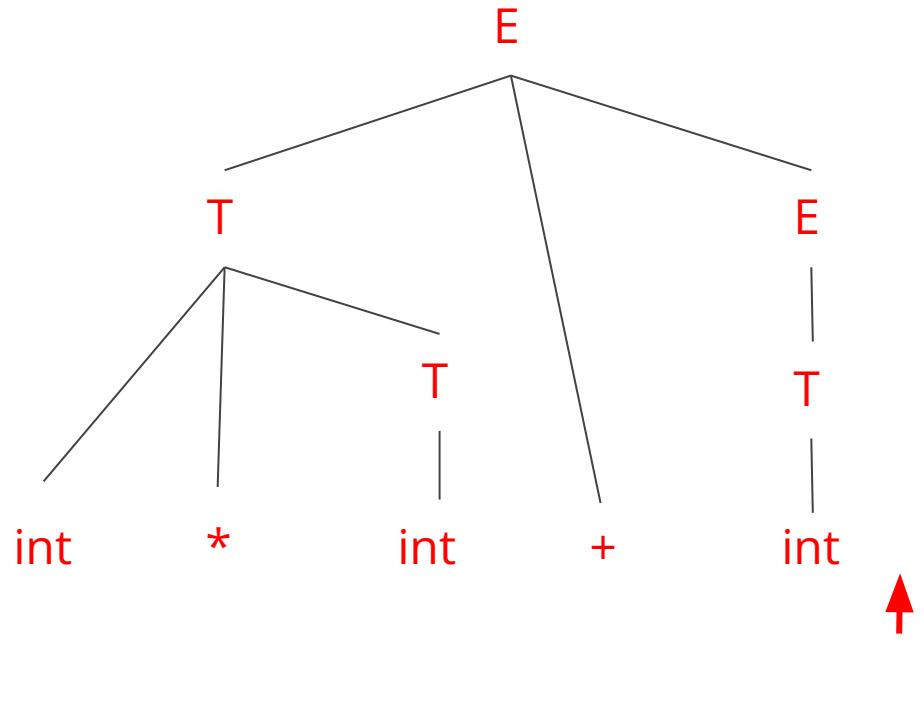
T + |int

T + int|

T + T|

T + E|

E|





PARSERS BOTTOM-UP

Considere la siguiente gramática y genere las operaciones Shift-Reduce para la cadena

- **(id+id)+id**

$E \rightarrow E' \mid E' + E$

$E' \rightarrow -E' \mid id \mid (E)$



PARSERS BOTTOM-UP

- La cadena izquierda puede ser implementada por una pila
- La parte superior de la pila es el |
- Shift empuja una terminal en la pila
- Reduce saca símbolos de la pila (producción derecha)
- Reduce coloca un no terminal en la pila (producción izquierda)



PARSERS BOTTOM-UP

- En un estado dado, más de una acción (cambiar o reducir) puede conducir a un análisis válido
- Si es legal cambiar o reducir, hay un conflicto de tipo *shift-reduce*
- Si es legal reducir en dos producciones diferentes, hay conflicto *reduce-reduce*

Handles



Handles

- ¿Cómo decidimos cuándo cambiar o reducir?
- Ejemplo de gramática:

$$\begin{aligned} E &\rightarrow T + E|T \\ T &\rightarrow \text{int}|\text{int} * T|(E) \end{aligned}$$

- Considere paso $\text{int}| * \text{int} + \text{int}$
- Podríamos reducir por $T \rightarrow \text{int}$ mediante $T| * \text{int} + \text{int}$
- Un error porque no hay forma de reducir al símbolo de inicio E



Handles

- Intuición: queremos reducir solo si el resultado aún puede ser reducido al símbolo de inicio
 - Supongamos una derivación más a la derecha
- $$S \rightarrow^* \alpha X \omega \rightarrow \alpha \beta \omega$$
- Entonces $\alpha \beta$ es un handle de $\alpha \beta \omega$



Handles

- Handles son la formalización de la intuición
 - Un handle es una reducción que también permite mediante más reducciones volver al símbolo de inicio
- Solo queremos reducir en handles



Handles

$$E \rightarrow E' | E + E$$

$$E' \rightarrow -E' | id | (E)$$

Dada la gramática a la derecha, identifica el handle para el siguiente estado del análisis shift-reduce:

$E' + -id | + -(id + id)$

- $E' + -id$
- id
- $-id$
- $E' + -E'$



Handles

Regla #2

En el análisis de shift-reduce, los handles aparecen solo en la parte superior de la pila, nunca dentro



Handles

Inducción informal sobre el número de movimientos de reducción:

- La pila está vacía inicialmente
- Inmediatamente después de reducir un handle
 - El no terminal más a la derecha en la parte superior de la pila
 - El siguiente handle debe estar a la derecha del no terminal más a la derecha porque esta es una derivación más a la derecha
 - La secuencia de shift me lleva al siguiente handle



Handles

- En el análisis shift-reduce, los handlers siempre aparecen en la parte superior de la pila
- Los handles nunca están a la izquierda del no terminal más a la derecha.
- Los algoritmos de análisis de abajo hacia arriba se basan en reconocer handles

Trabajando con Handles

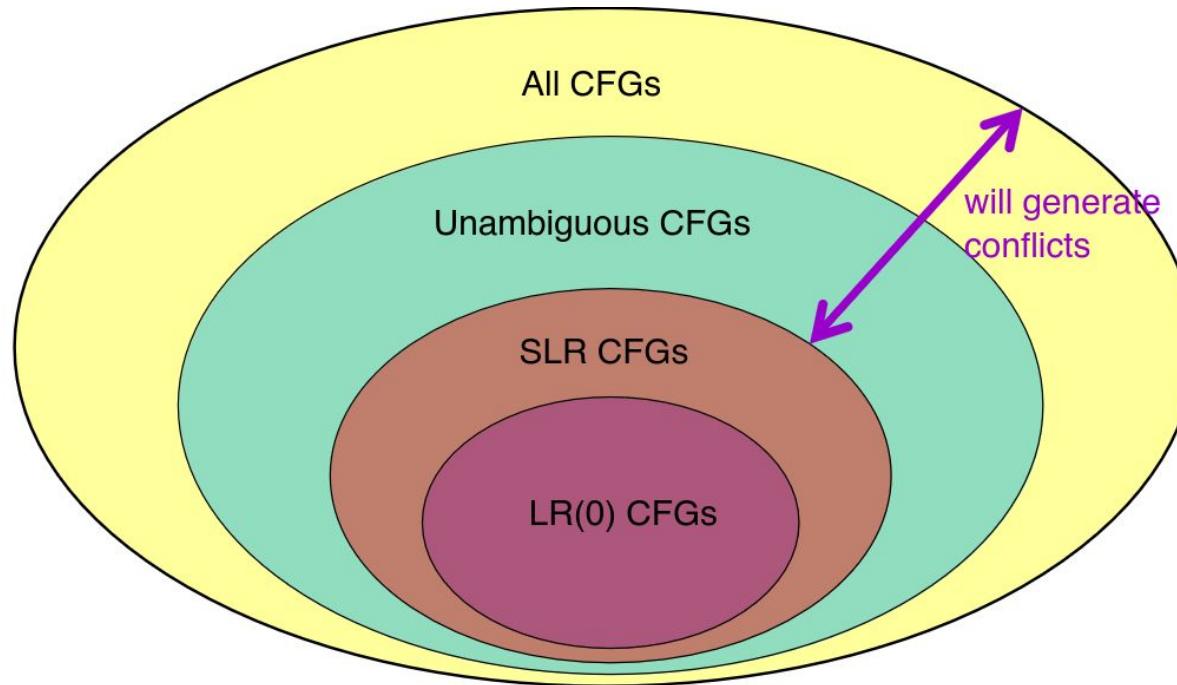


Trabajando con Handles

- Mala noticias: No existen algoritmos eficientes para reconocer Handles
- Buenas noticias:
 - Hay buenas heurísticas para adivinar handles
 - En algunos CFG, las heurísticas siempre adivinan



Trabajando con Handles





Trabajando con Handles

- No es obvio cómo detectar Handles
- En cada paso, el analizador solo ve la pila, no la entrada completa
- α es un prefijo viable si hay una ω tal que $\alpha|\omega$ es un estado de un analizador shift-reduce



Trabajando con Handles

- Un prefijo viable no se extiende más allá del extremo derecho del Handle
- Es un prefijo viable porque es un prefijo del Handle
- Mientras un analizador tenga prefijos viables en la pila no se ha detectado ningún error de análisis



Trabajando con Handles

Regla #3

Para cualquier gramática, el conjunto de prefijos viables es un lenguaje regular



Trabajando con Handles

Items:

- Un ítem es una producción con un “.” en algún lugar del lado derecho de la producción
- Los ítems para $T \rightarrow (E)$ son:
 - $T \rightarrow .(E)$
 - $T \rightarrow (.E)$
 - $T \rightarrow (E.)$
 - $T \rightarrow (E).$



Trabajando con Handles

- El problema de reconocer prefijos viables es que la pila tiene sólo partes y piezas del lado derecho de la producción
- Si estuviera completo, podríamos reducir
- Estos bits y piezas son siempre prefijos de producciones en el lado derecho



Trabajando con Handles

Considere la entrada: **(int)**
para la gramática:

$$E \rightarrow T + E|T$$

$$T \rightarrow \text{int} * T|\text{int}|(E)$$

- Entonces **(E|)** es un estado de un análisis shift-reduce
- **(E** es un prefijo del lado derecho de la producción $T \rightarrow (E)$ que se reducirá (reduce) al siguiente shift
- El ítem $T \rightarrow (E.)$ dice que hasta ahora hemos visto **(E** de esta producción y esperamos ver **)**



Trabajando con Handles

- La pila puede tener muchos prefijos de lados derechos de producciones:
 $\text{Prefix}_1 \text{ Prefix}_2 \dots \text{Prefix}_{n-1} \text{ Prefix}_n$
- Sea Prefix_i un prefijo el lado derecho de la producción $X_i \rightarrow a_i$
 - Prefix_i eventualmente se reducirá a X_i
 - La parte que falta de a_{i-1} comienza con X_i
 - Es decir, hay un $X_{i-1} \rightarrow \text{Prefix}_{i-1} X_i \beta$ para algún β
- De forma recursiva, $\text{Prefix}_{k+1} \dots \text{Prefix}_n$ finalmente se reduce a la parte que falta de a_k



Trabajando con Handles

La “pila de ítems”

$$T \rightarrow (\cdot E)$$

$$E \rightarrow \cdot T$$

$$T \rightarrow \text{int} * \cdot T$$

Significa:

- Hemos visto “(\cdot ” de $T \rightarrow (\cdot E)$
- Hemos visto ϵ de $E \rightarrow \cdot T$
- Hemos visto $\text{int}*$ de $T \rightarrow \text{int} * \cdot T$



Trabajando con Handles

Idea: Para reconocer prefijos viables, debemos

- Reconocer una secuencia de lados derechos de producciones parciales donde ...
- Cada lado derecho parcial puede eventualmente reducirse a una parte de el sufijo faltante de su predecesor

Reconociendo Prefixes Viabiles



Reconociendo Prefixes Viables

1. Añadir una producción ficticia $S' \rightarrow S$ hacia G
2. Los estados NFA son los elementos de G
3. Para el ítem $E \rightarrow \alpha.X\beta$, agregue la transición $E \rightarrow \alpha.X\beta \xrightarrow{x} E \rightarrow \alpha X.\beta$
4. Para el ítem $E \rightarrow \alpha.X\beta$ y la producción $X \rightarrow \Delta$, agregue
5. $E \rightarrow \alpha.X\beta \xrightarrow{\epsilon} X \rightarrow .\Delta$
6. Cualquier estado es un **estado de aceptación**
7. El estado inicial es $S' \rightarrow .S$



Reconociendo Prefixes Viables

$S \rightarrow E$

$E \rightarrow T + E|T$

$T \rightarrow \text{int} * T|\text{int}|(E)$

Definimos el NFA...



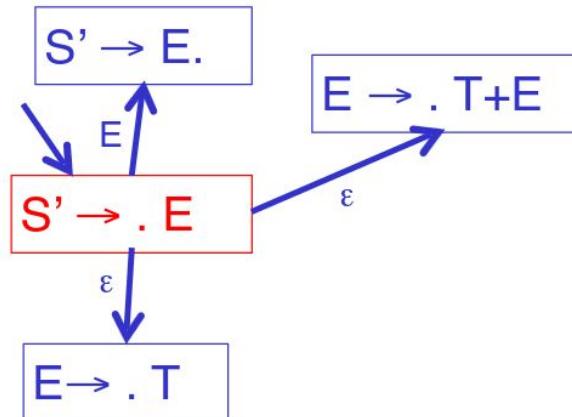
Reconociendo Prefixes Viables

$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} \mid (E) \end{aligned}$$

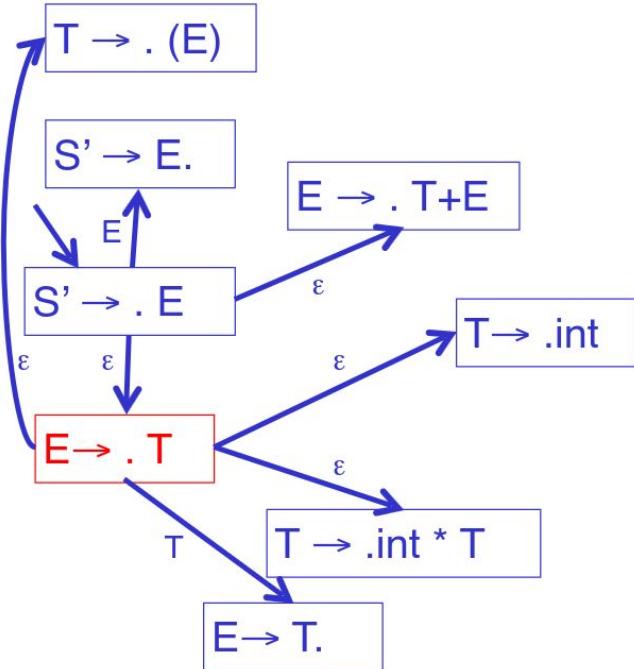
$S' \rightarrow . E$



Reconociendo Prefixes Viables

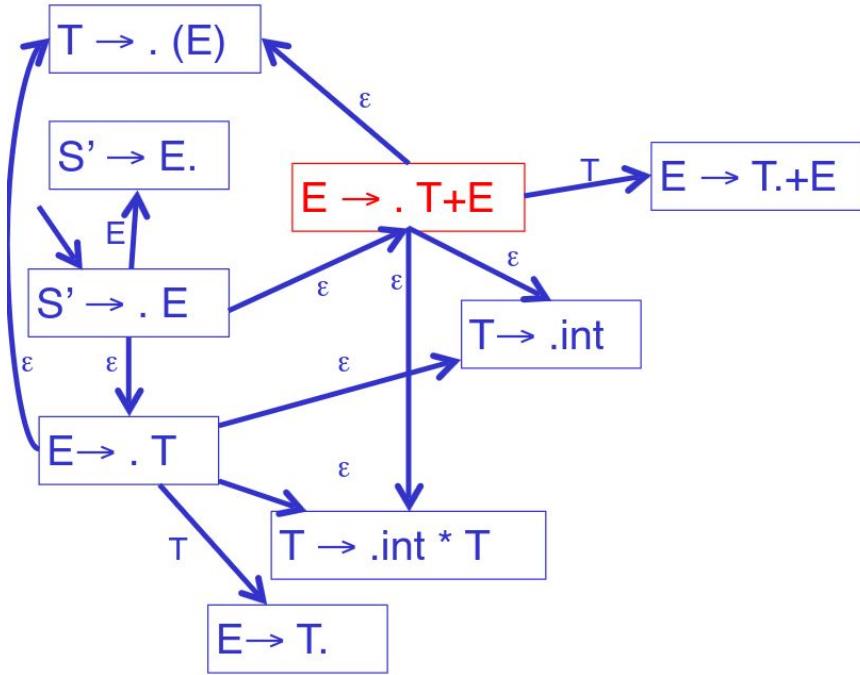
$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} \mid (E) \end{aligned}$$


Reconociendo Prefixes Viables



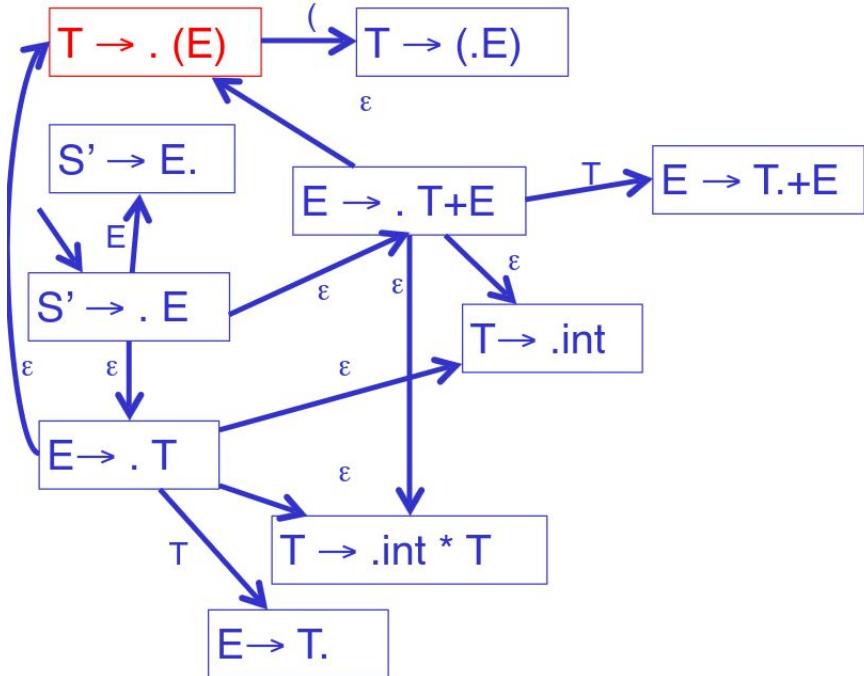
$E \rightarrow T + E | T$
 $T \rightarrow \text{int} * T | \text{int} | (E)$

Reconociendo Prefixes Viables

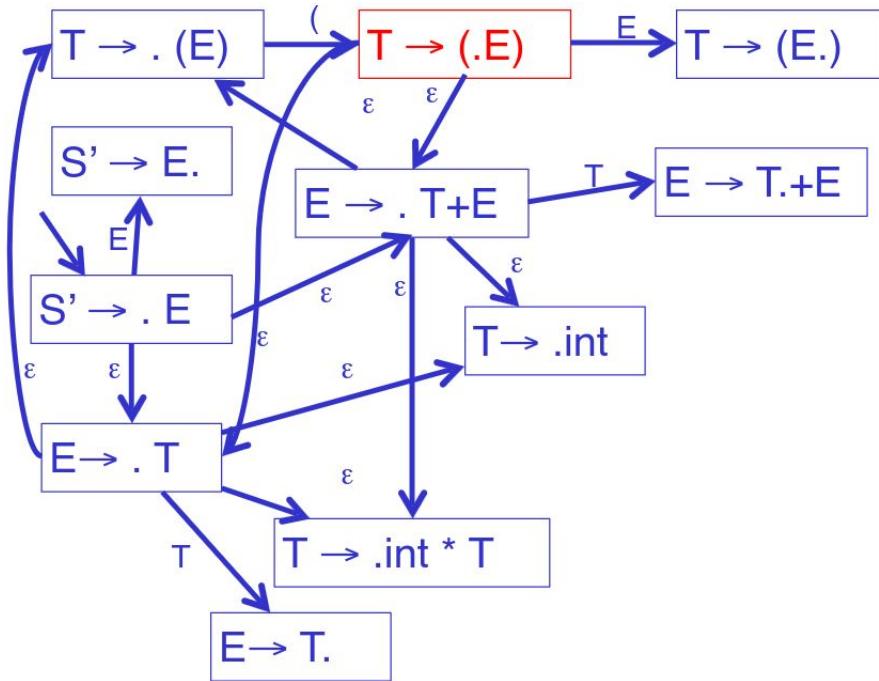


$E \rightarrow T + E \mid T$
 $T \rightarrow \text{int} * T \mid \text{int} \mid (E)$

Reconociendo Prefixes Viables

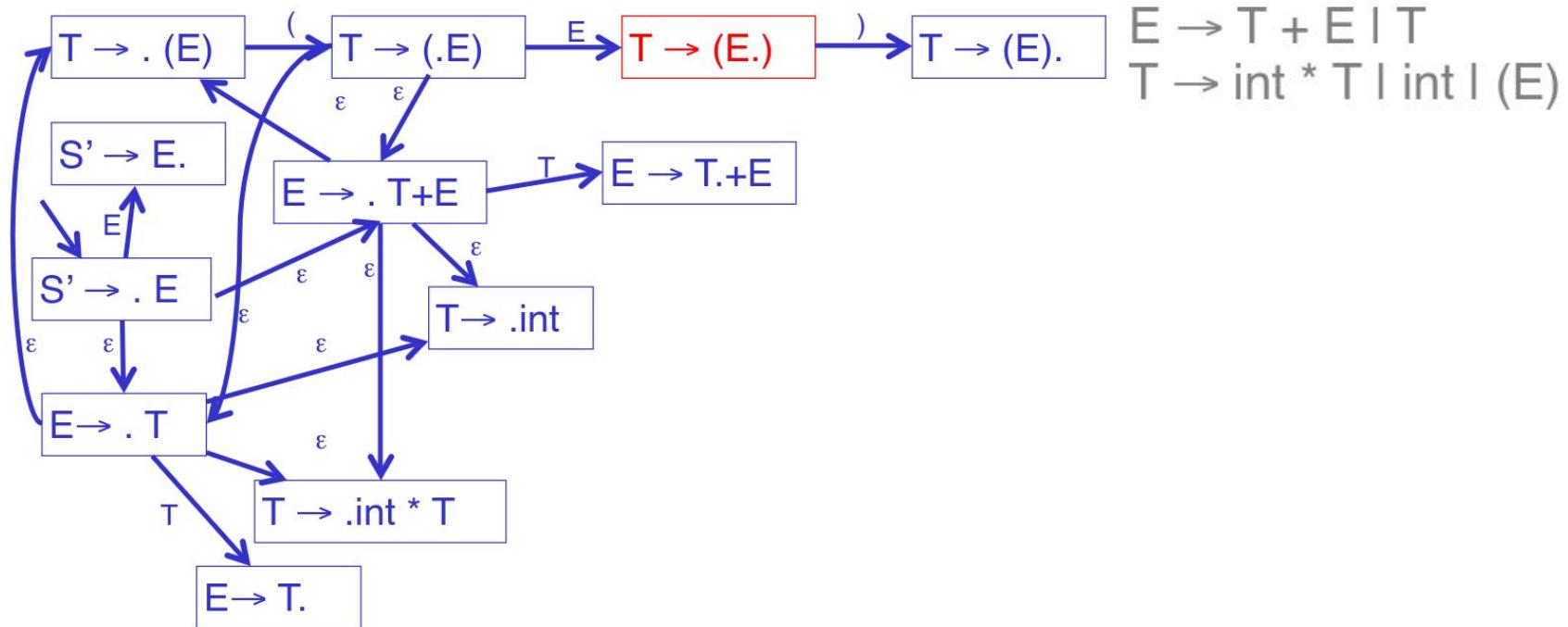

$$\begin{aligned}E &\rightarrow T + E \mid T \\T &\rightarrow \text{int}^* T \mid \text{int} \mid (E)\end{aligned}$$

Reconociendo Prefixes Viables

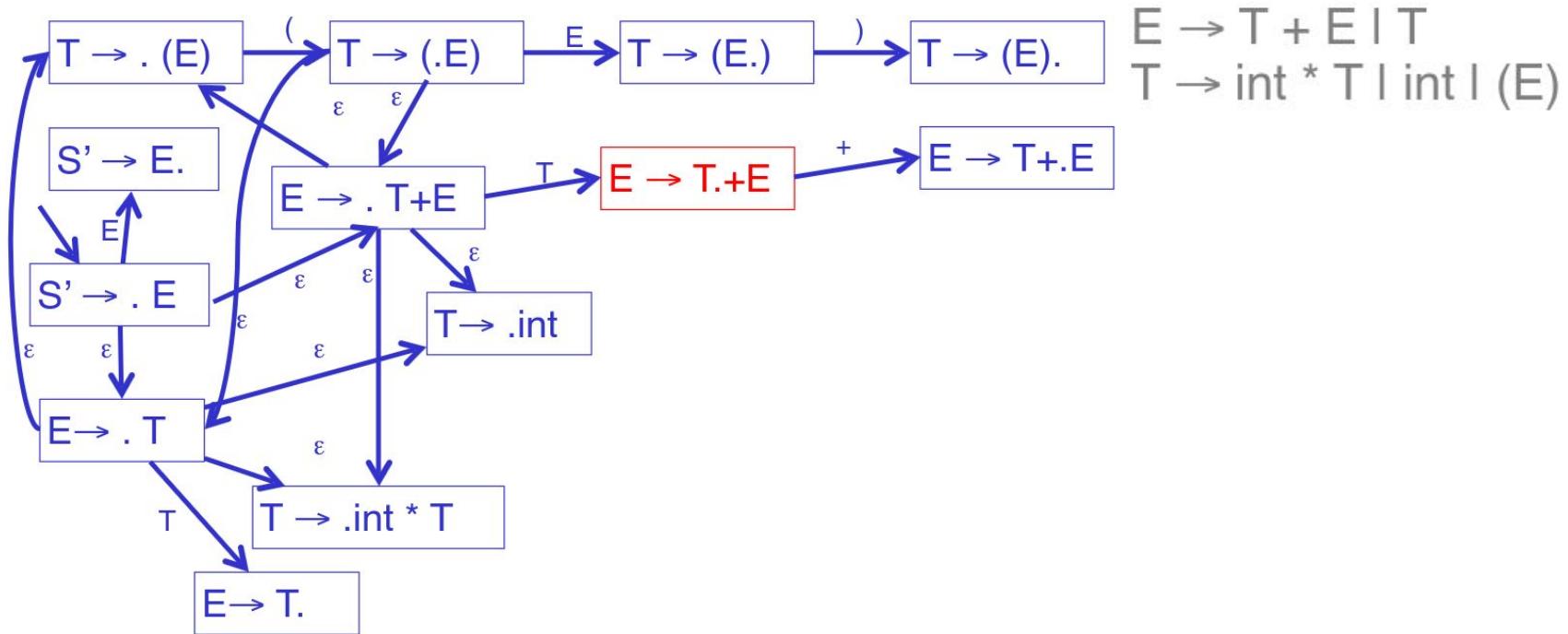


$E \rightarrow T + E \mid T$
 $T \rightarrow \text{int} * T \mid \text{int} \mid (E)$

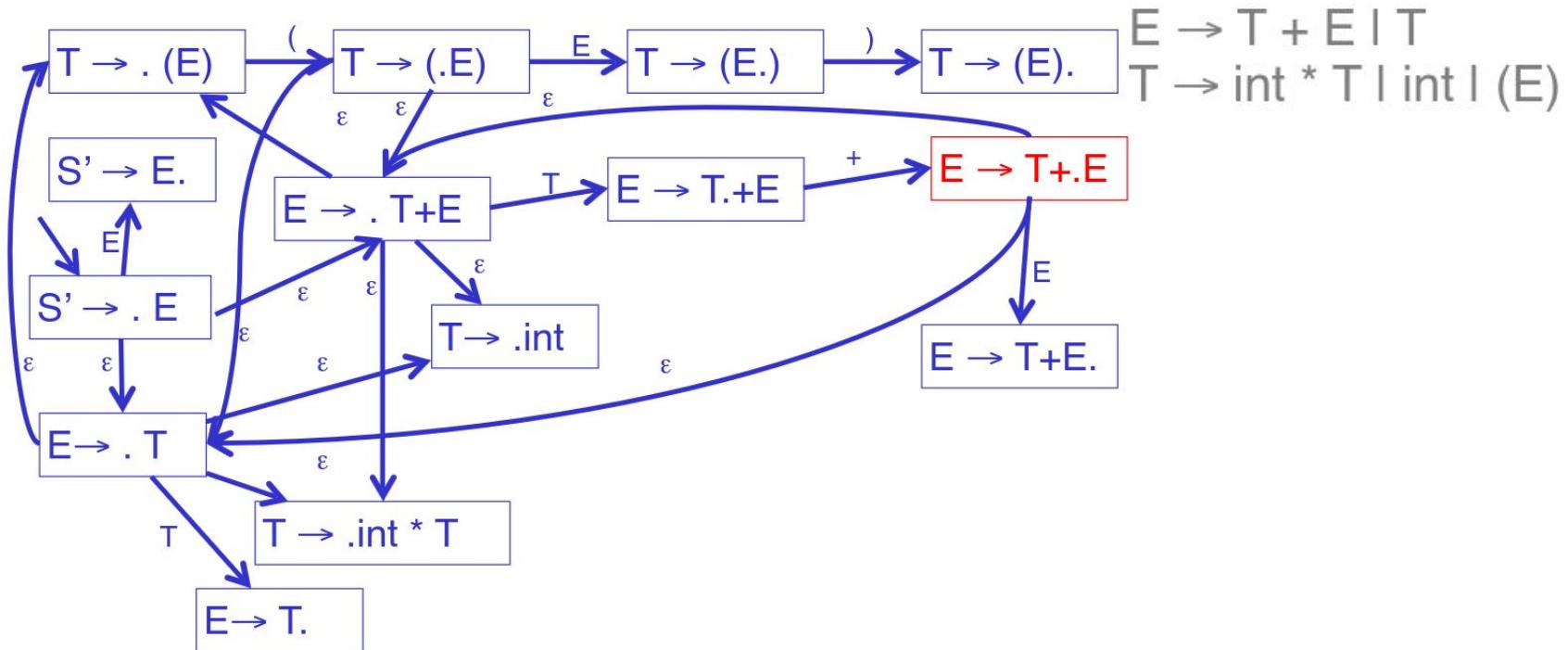
Reconociendo Prefixes Viables



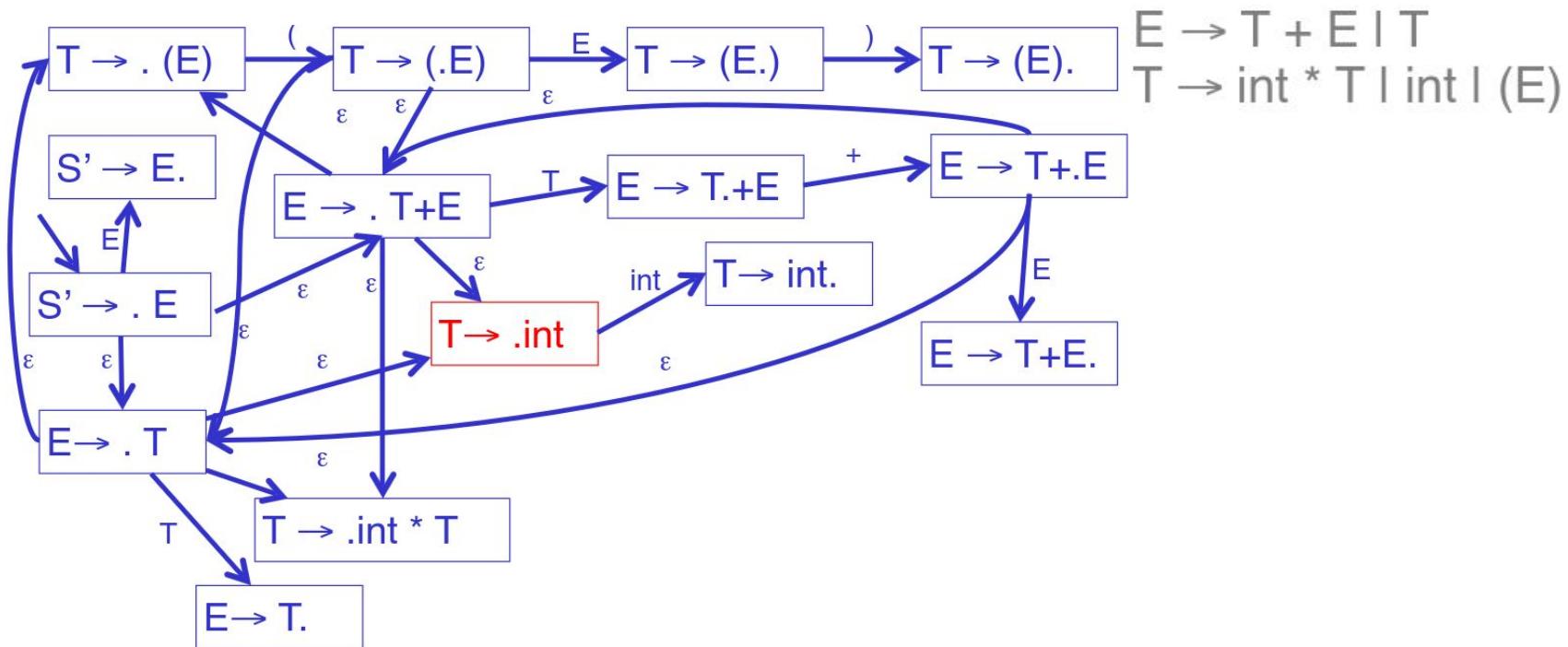
Reconociendo Prefixes Viables



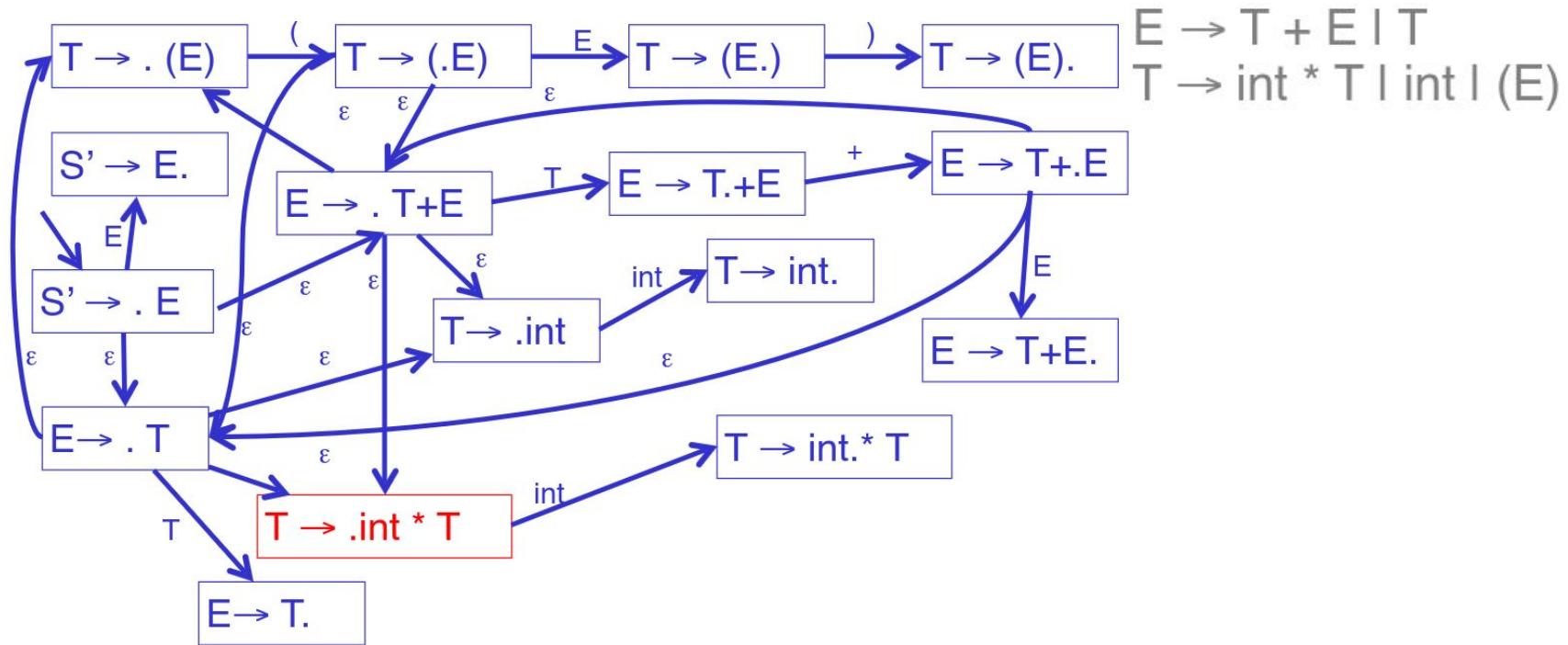
Reconociendo Prefixes Viables



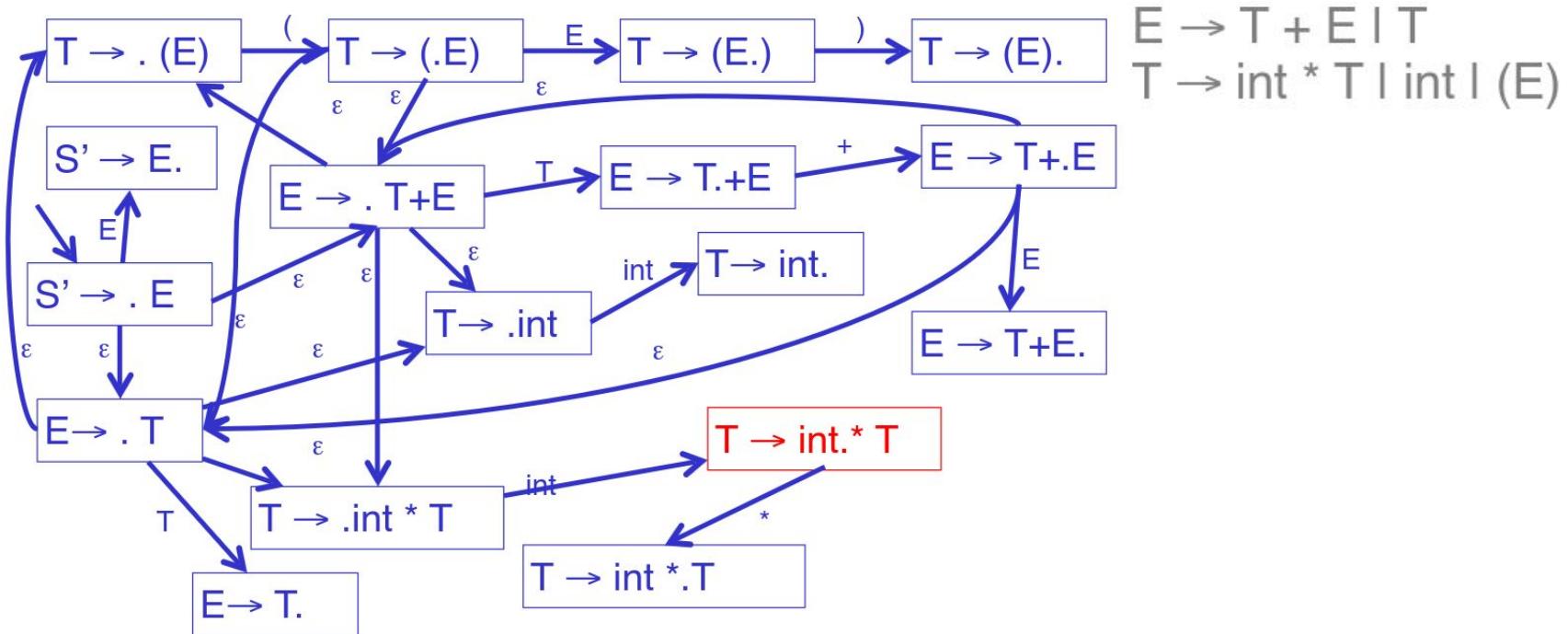
Reconociendo Prefixes Viables



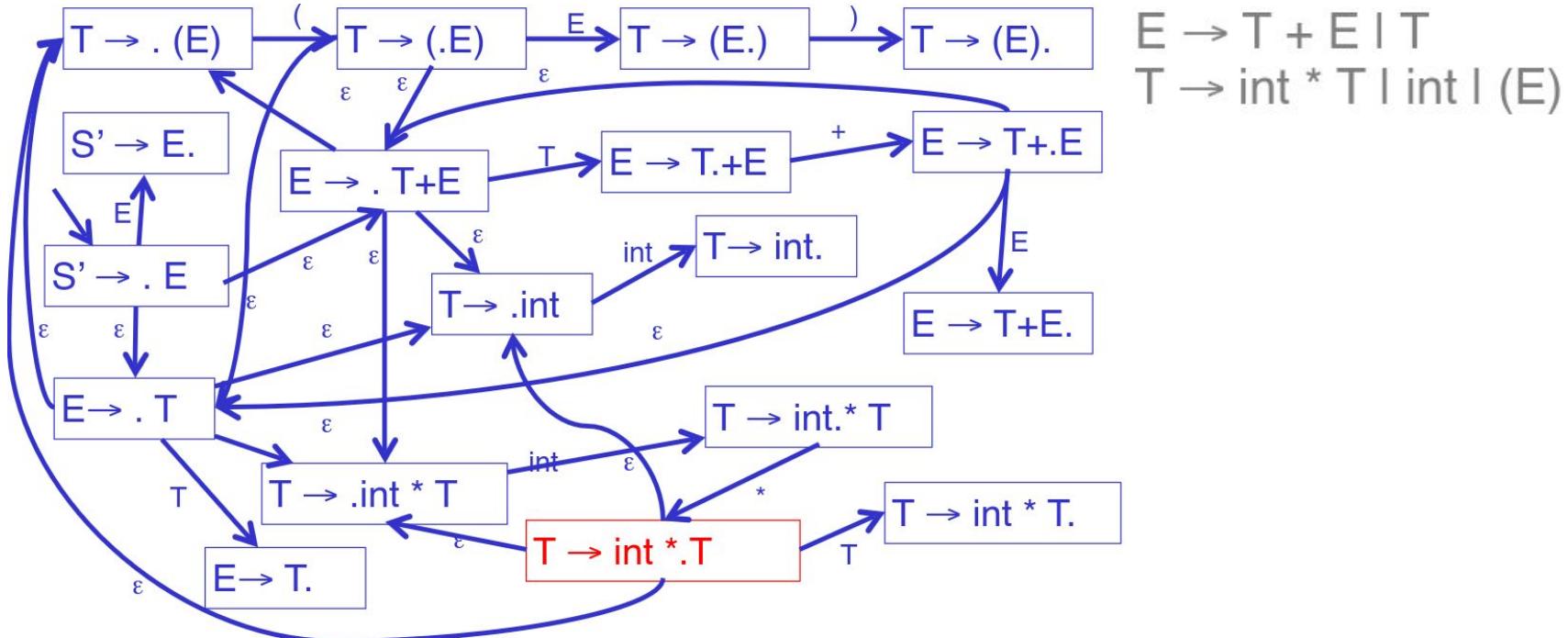
Reconociendo Prefixes Viables



Reconociendo Prefixes Viables

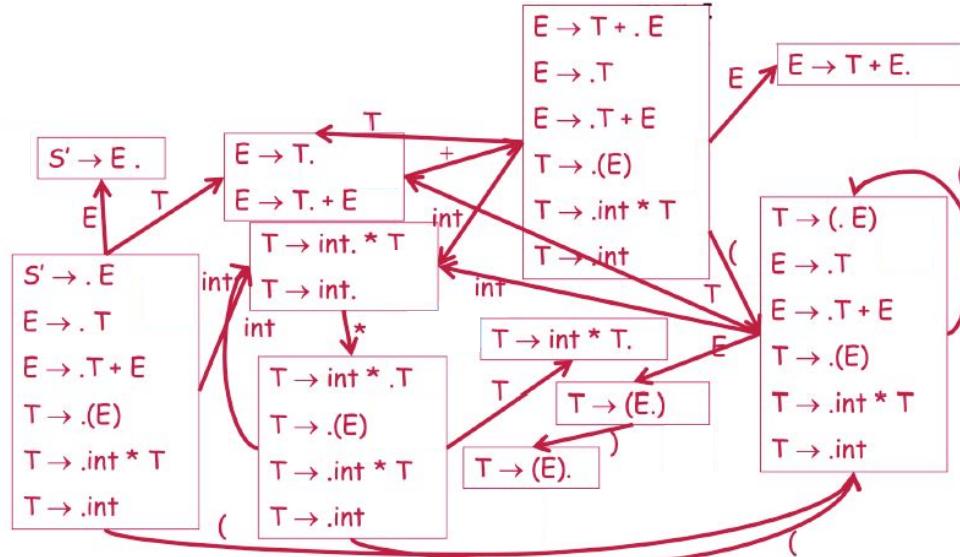


Reconociendo Prefixes Viables



Reconociendo Prefixes Viables

Convirtiendo en
DFA





Reconociendo Prefixes Viables

Ítem Válidos

El ítem $X \rightarrow \beta.y$ es válido para un prefijo viable $\alpha\beta$ si

$$S' \xrightarrow{*} \alpha X \omega \rightarrow \alpha \beta y \omega$$

para una derivación por la derecha.

Después de analizar $\alpha\beta$, los elementos válidos son las posibles partes superiores de la pila de elementos.



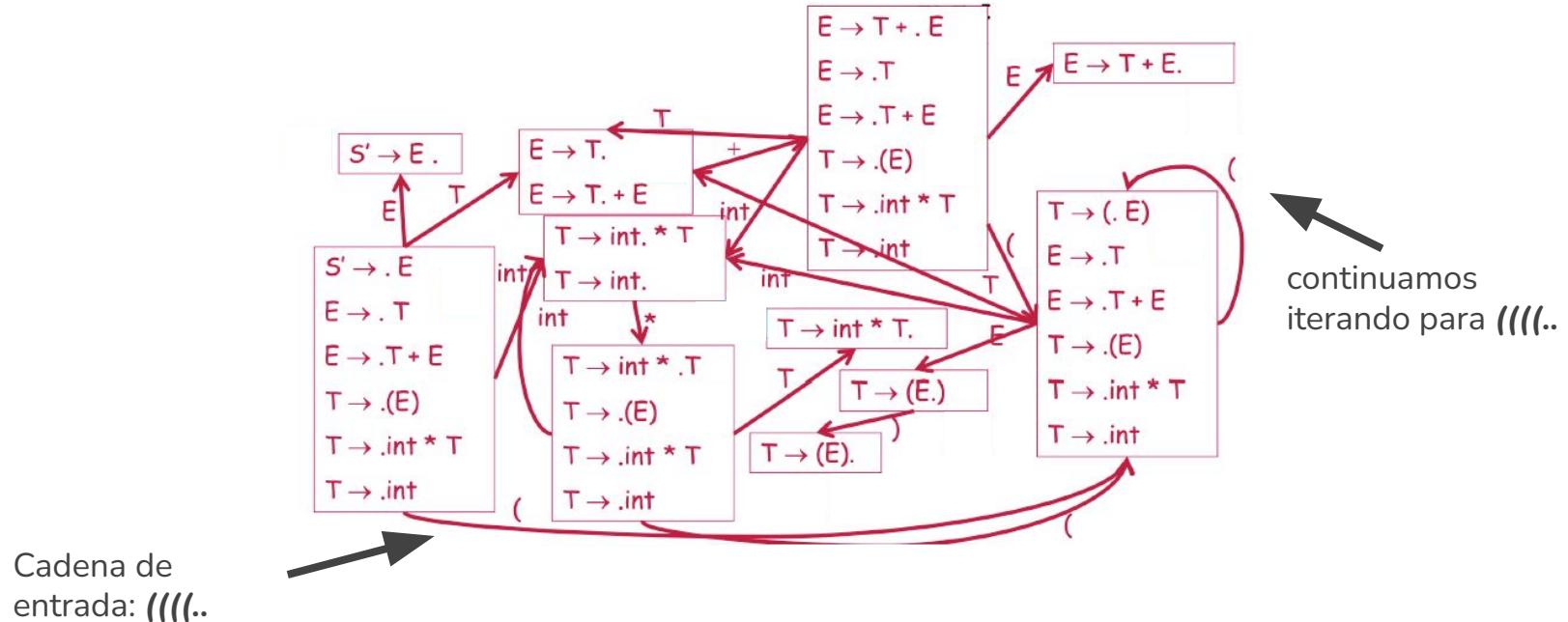
Reconociendo Prefixes Viables

Un ítem suele ser válido para muchos prefijos.

Por ejemplo: El ítem $T \rightarrow (\textcolor{red}{.}E)$ es válido para los prefijos:

(
((
(((
((((
...
..

Reconociendo Prefixes Viables



Parsing LR(0) y SLR



Parsing LR(0)

Idea: Supongamos que

- el stack contiene α
- el siguiente símbolo de entrada es t
- el DFA con la entrada α termina en el estado s
- Reducir por $X \rightarrow \beta$ si
 - s contiene el ítem $X \rightarrow \beta$.
- Hacer shift si
 - s contiene el ítem $X \rightarrow \beta \cdot t \omega$
 - es equivalente a decir que s tiene una transición etiquetada con t .



Parsing LR(0)

Conflictos

LR(0) tiene un conflicto de reduce/reduce si:

- Cualquier estado tiene dos ítems de reduce:
 - $X \rightarrow \beta.$ y $Y \rightarrow \omega.$

LR(0) tiene un conflicto de shift/reduce si:

- Cualquier estado tiene un ítem de reduce y un ítem de shift:
 - $X \rightarrow \beta.$ y $Y \rightarrow \omega . t \delta$



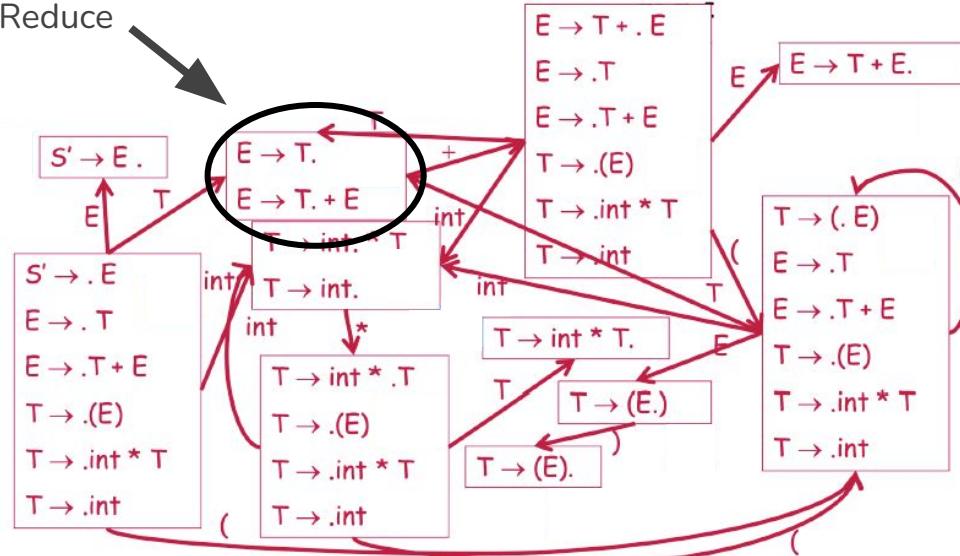
Parsing LR(0)

3) El DFA termina en un estado s :

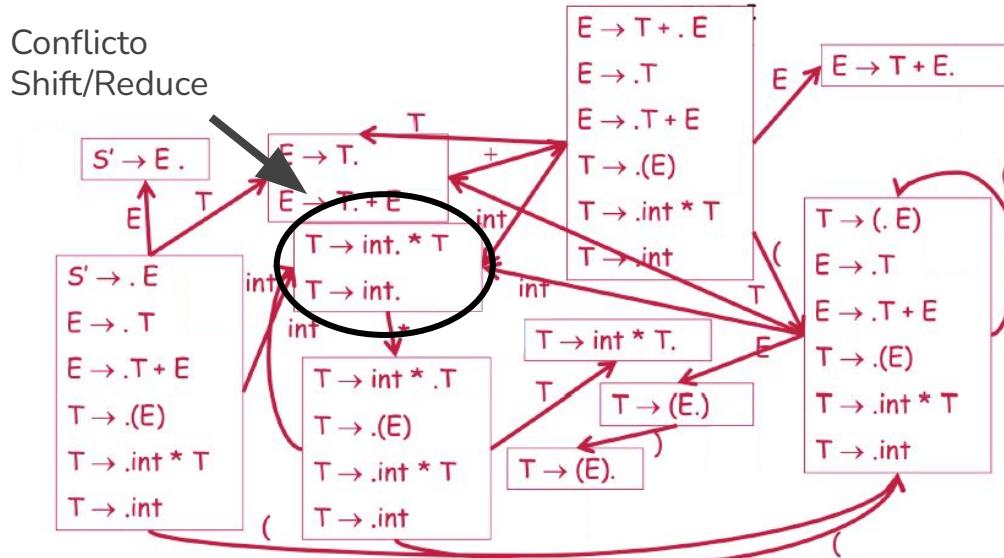
- El DFA ha procesado el prefijo α de la cadena de entrada y ha llegado al estado s .
- Este estado s contiene un conjunto de ítems que describen qué producciones de la gramática son posibles dado el prefijo α .

Reconociendo Prefixes Viables

Conflict Shift/Reduce



Reconociendo Prefixes Viables





SLR

LR = “Left-to-right scan”

SLR = “Simple LR”

SLR mejora las heurísticas de shift/reduce de LR(0)

- Menos estados tienen conflictos



Parsing SLR

Idea: Supongamos que

- el stack contiene α
- el siguiente símbolo de entrada es t
- el DFA con la entrada α termina en el estado s
- Reducir por $X \rightarrow \beta$ si
 - s contiene el ítem $X \rightarrow \beta$.
 - $t \in Follow(X)$
- Hacer shift si
 - s contiene el ítem $X \rightarrow \beta \cdot t \omega$



Parsing SLR

Si hay conflictos bajo estas reglas, la gramática no es SLR.

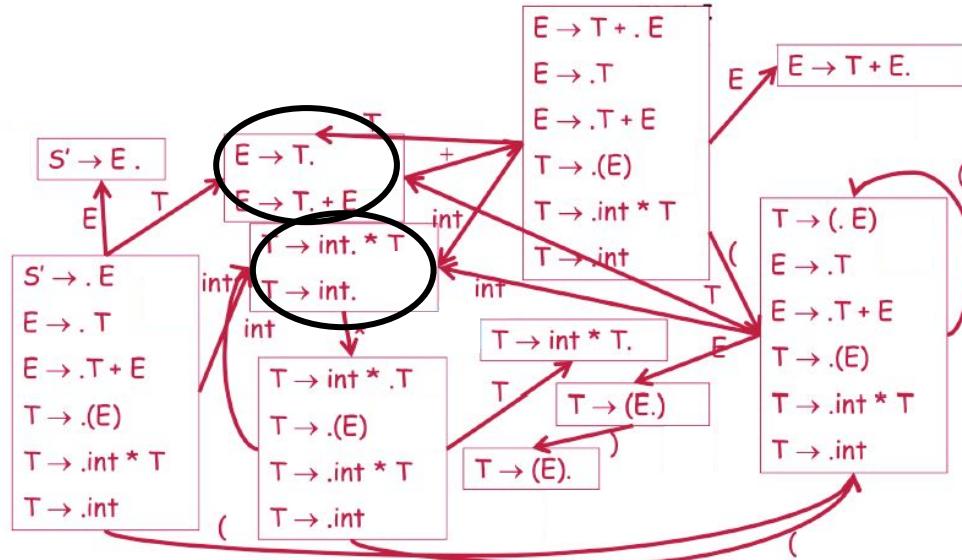
Las reglas equivalen a una heurística para detectar handles

- Las gramáticas SLR son aquellas donde las heurísticas detectan exactamente los handles.



Parsing SLR

$$\text{Follow}(E) = \{ ')', \$ \}$$
$$\text{Follow}(T) = \{ '+', ')', \$ \}$$





Parsing SLR

- Muchas de las gramáticas no son SLR
 - incluidas todas las gramáticas ambiguas
- Podemos analizar más gramáticas mediante el uso de declaraciones de precedencia
- - Instrucciones para resolver conflictos



Parsing SLR

Considera nuestra gramática ambigua favorita:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{int}$$

El DFA para esta gramática contiene un estado con los siguientes ítems:

- $E \rightarrow E * E .$
- $E \rightarrow E . + E$

¡conflicto de shift/reduce!

Declarar "** tiene mayor precedencia que +*" resuelve este conflicto a favor de reducir.



Parsing SLR

El término "declaración de precedencia" es **engañoso**.

Estas declaraciones no definen la precedencia; definen **resoluciones de conflictos**.

¡No es exactamente lo mismo!



Algoritmo SLR sin optimizar

Sea M el DFA para los prefijos viables de G .

Sea $|x_1 \dots x_n \$|$ la configuración inicial.

Repetir hasta que la configuración sea $S|\$$:

- Sea $\alpha|\omega$ la configuración actual.
- Ejecutar M en el stack actual α .
- Si M rechaza α , reportar error de análisis.
 - El stack α no es un prefijo viable.
- Si M acepta α con ítems L , sea t la siguiente entrada.
 - Reduce si $X \rightarrow \beta. \in L$ y $t \in \text{Follow}(X)$
 - De lo contrario, shift si $X \rightarrow \beta. t \gamma \in L$
 - Reportar error de análisis si ninguna de las dos se aplica.



Parsing SLR

- Si hay un conflicto en el último paso, la gramática no es SLR(k).
- k es la cantidad de anticipación.
 - En la práctica, $k = 1$.
- Se omitirá el uso de un estado inicial adicional S' en el siguiente ejemplo para ahorrar espacio en las diapositivas.



Parsing SLR

Gramática

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

Entrada

$$\text{int} * \text{int}$$

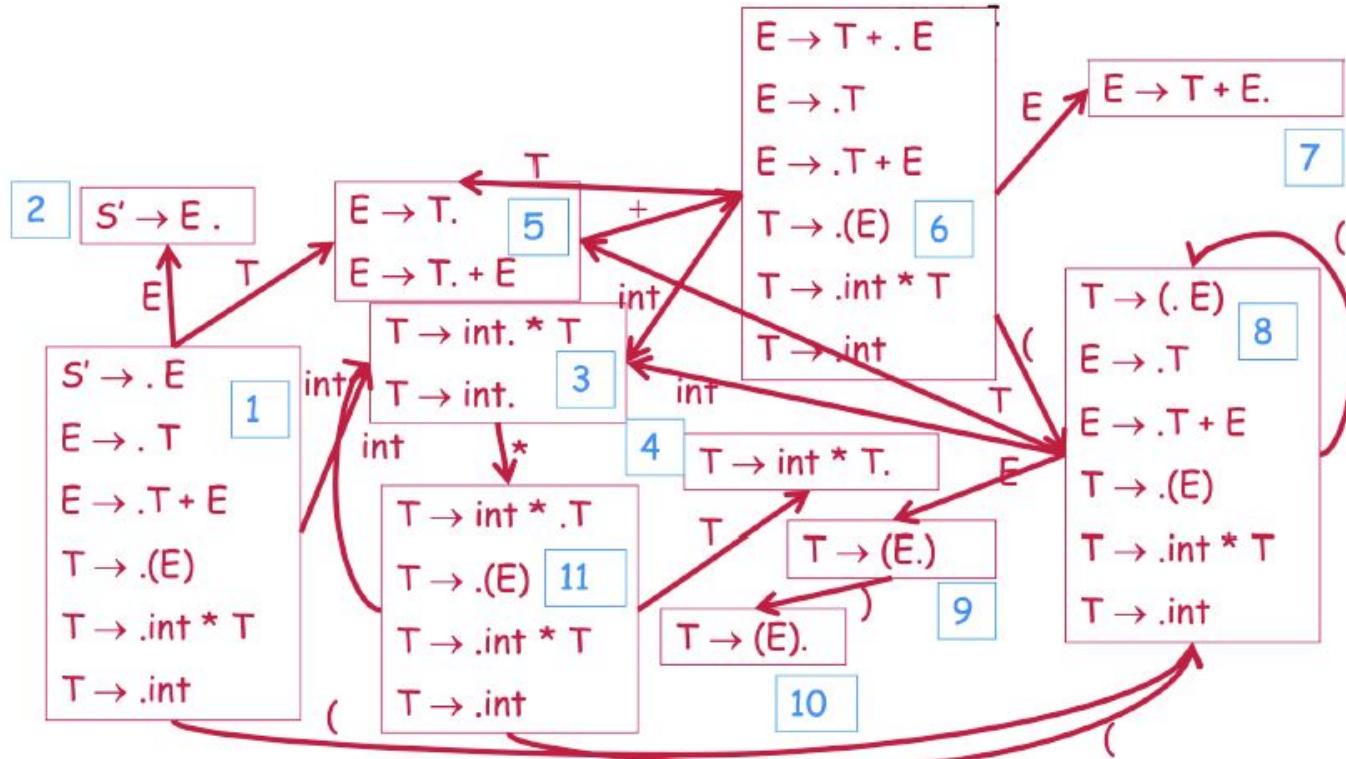
Símbolo de aceptación

\$

Parsing SLR

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$



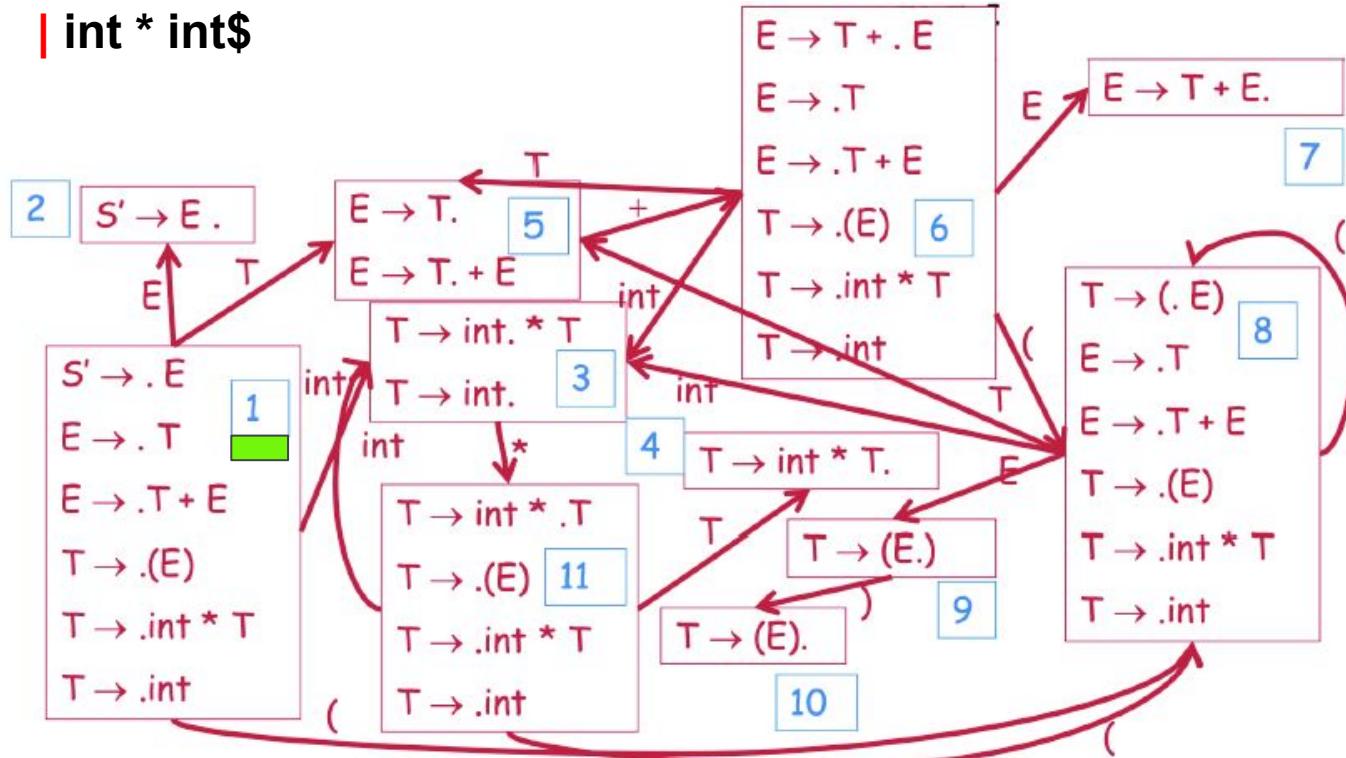
Parsing SLR

$$\text{Follow}(E) = \{ ')', \$ \}$$

Parsing SLR

$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} \mid (E) \end{aligned}$$

| int * int\$





Parsing SLR

$\text{Follow}(E) = \{ ')', \$ \}$
 $\text{Follow}(T) = \{ '+', ')', \$ \}$

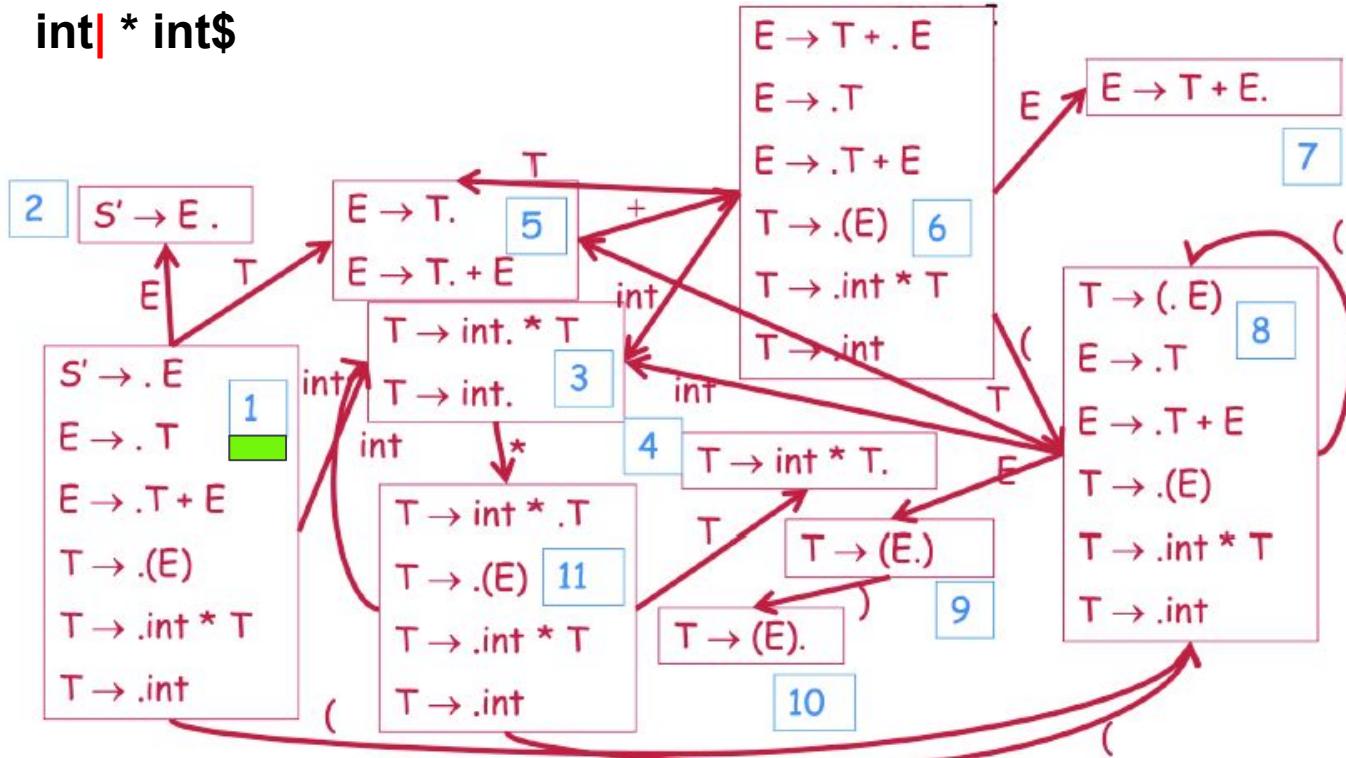
Entrada	Estado DFA	Acción
int * int\$	1	shift
int * int\$	3 * no es follow de T	shift

Parsing SLR

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

int| * int\$

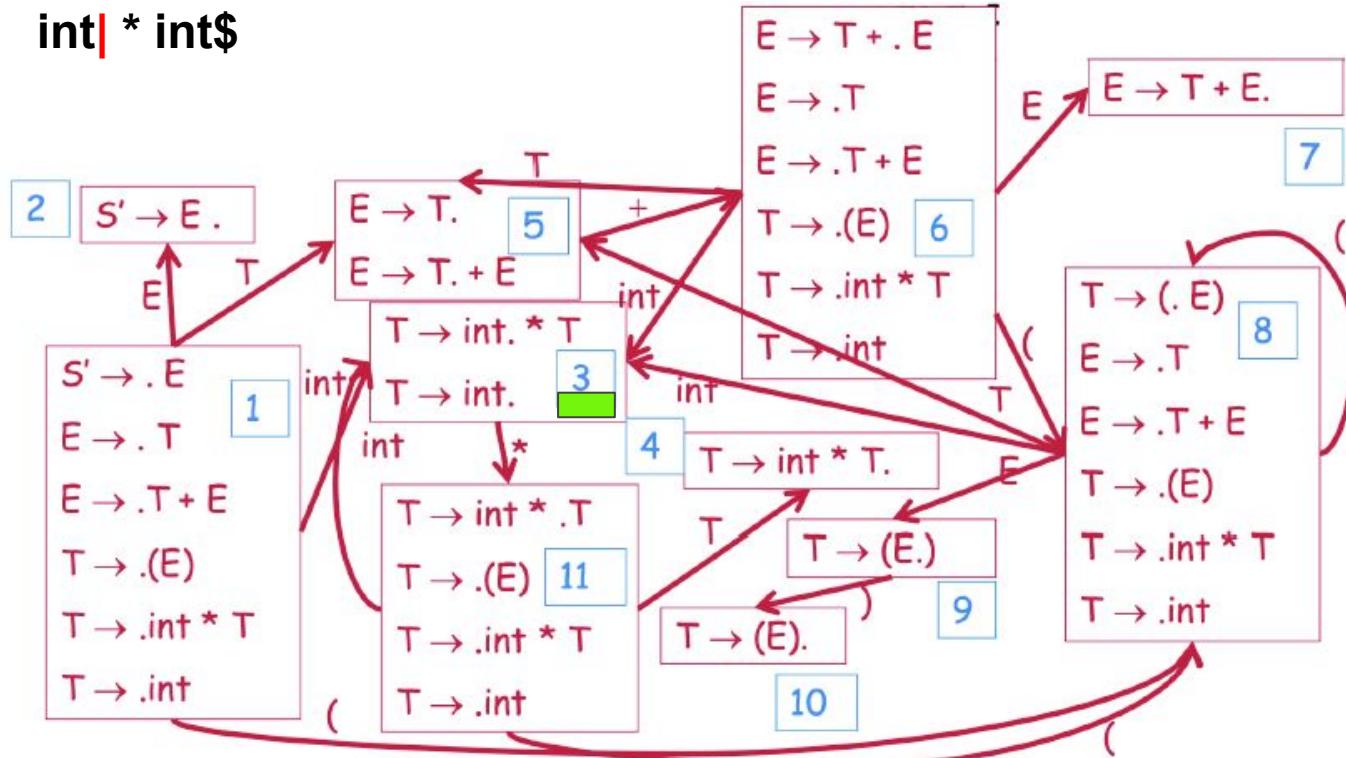


Parsing SLR

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

int| * int\$





Parsing SLR

Follow(E) = { ')', '\$' }
Follow(T) = { '+', ')', '\$' }

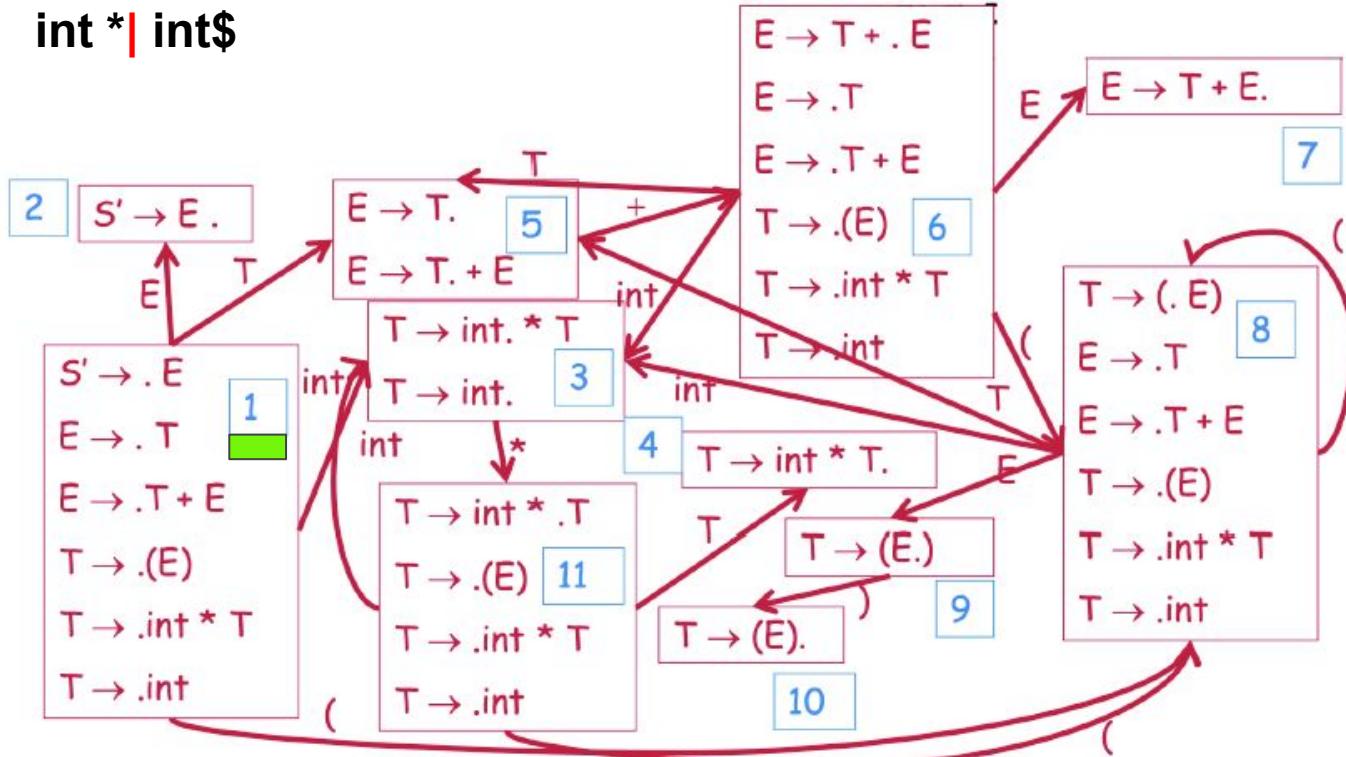
Entrada	Estado DFA	Acción
int * int\$	1	shift
int * int\$	3 * no es follow de T	shift
int * int\$	11	shift

Parsing SLR

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

int * | int\$

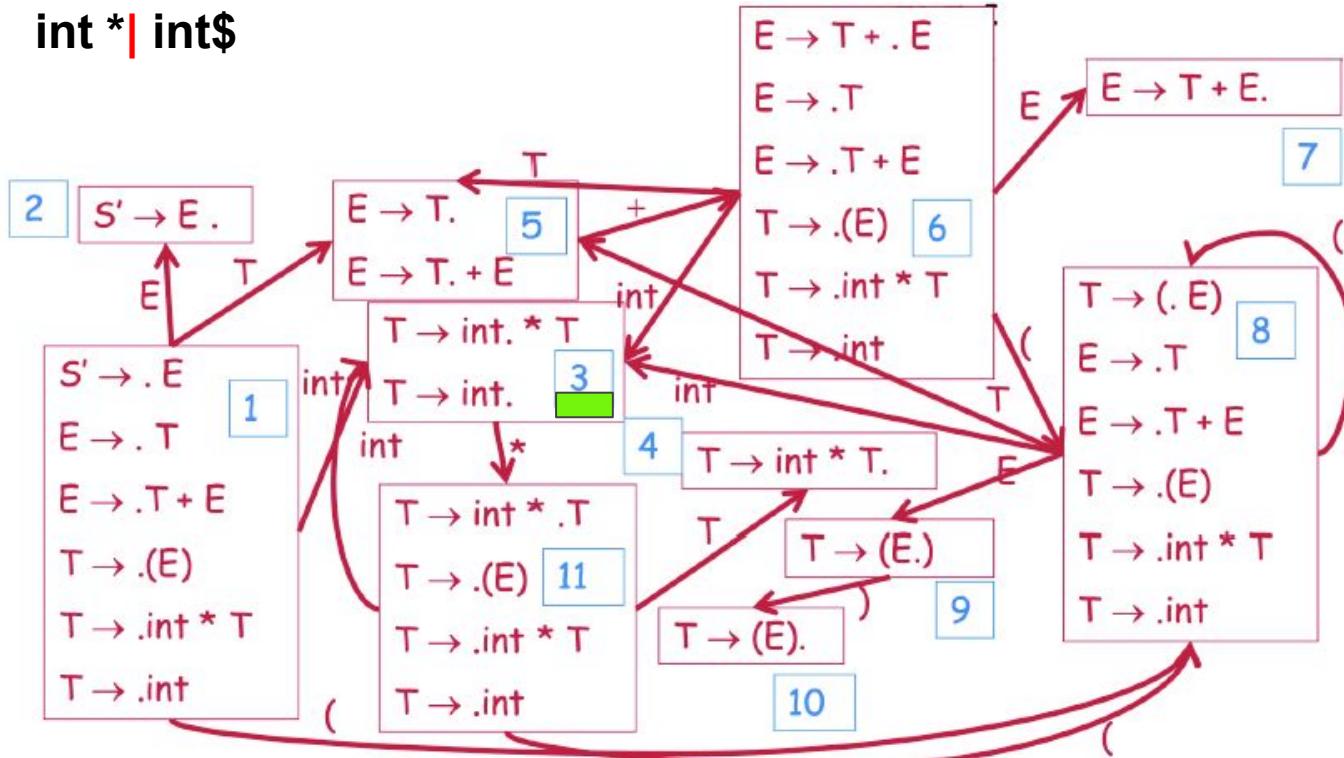


Parsing SLR

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

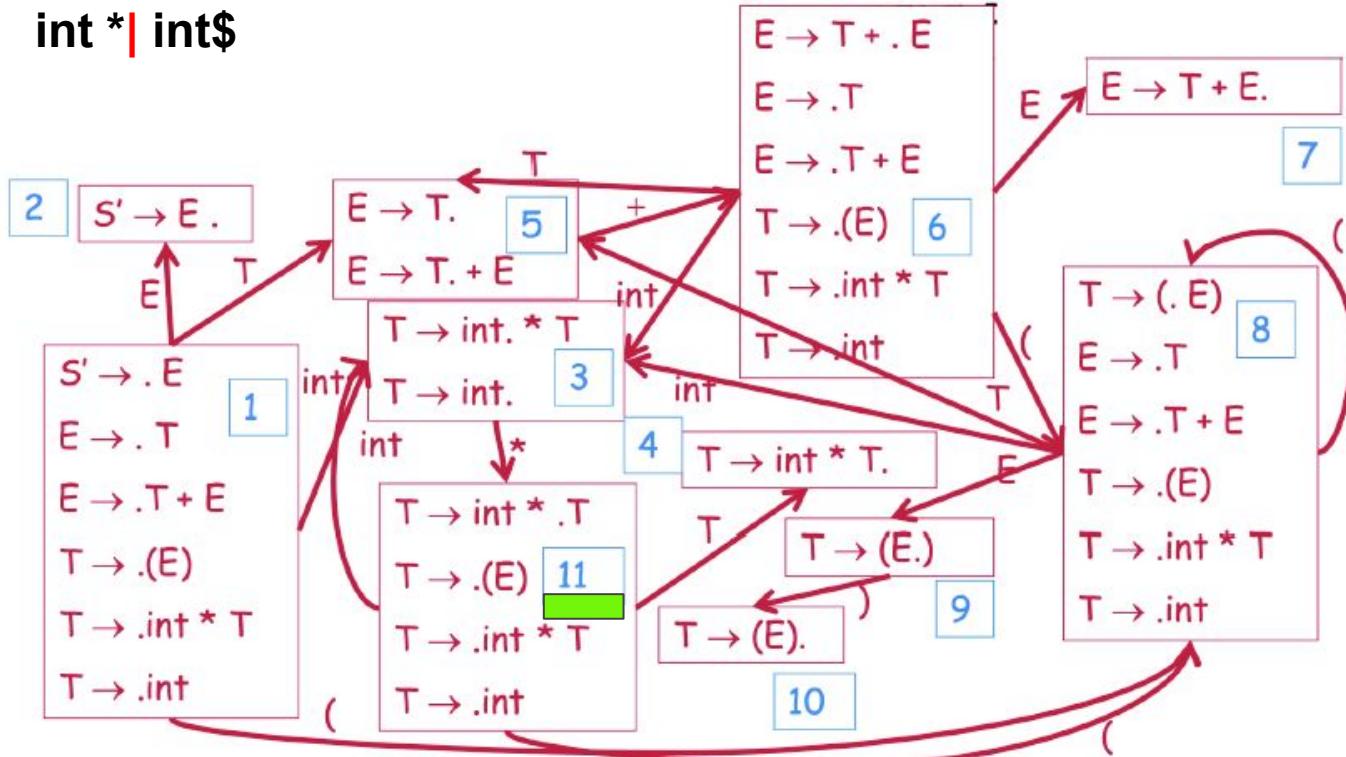
int * | int\$



Parsing SLR

$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} \mid (E) \end{aligned}$$

int * | int\$





Parsing SLR

$\text{Follow}(E) = \{ ')', \$ \}$
 $\text{Follow}(T) = \{ '+', ')', \$ \}$

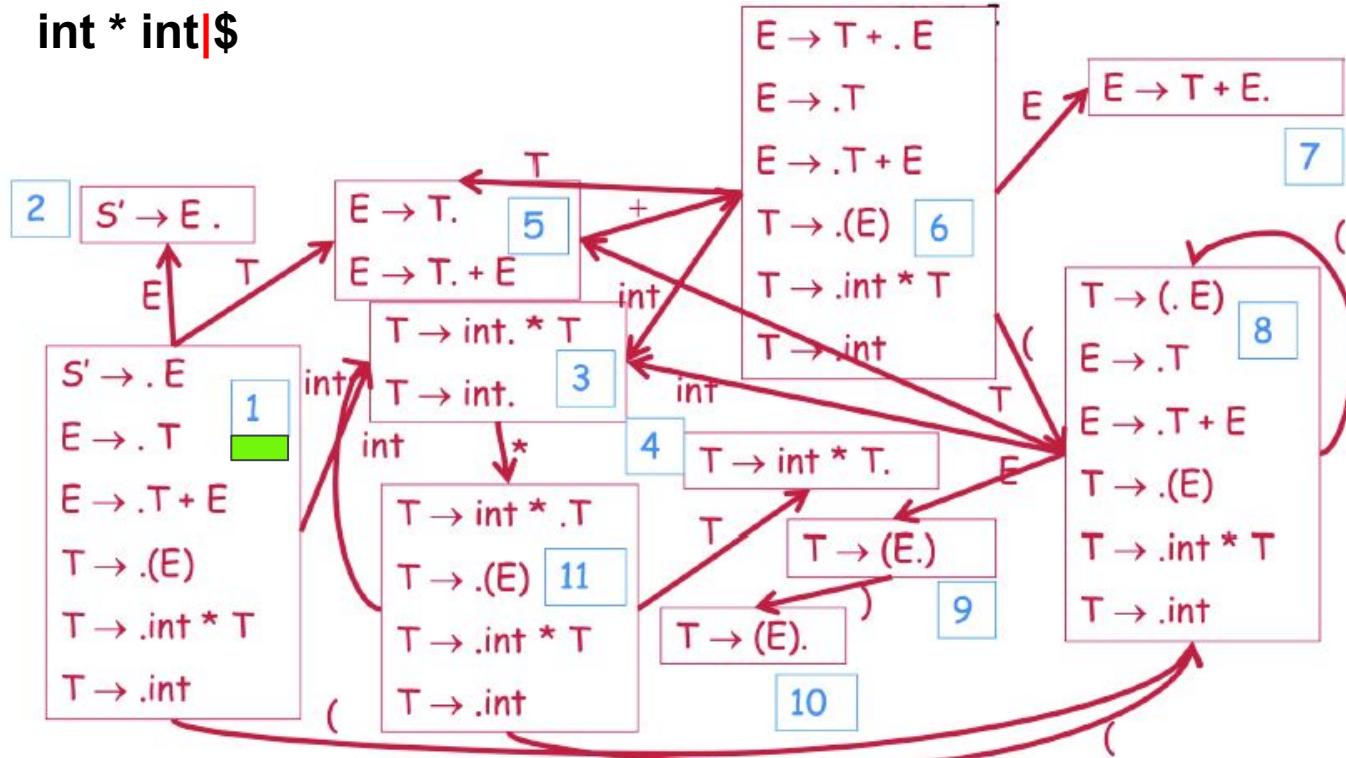
Entrada	Estado DFA	Acción
int * int\$	1	shift
int * int\$	3 * no es follow de T	shift
int * int\$	11	shift
int * int \$	\$ ∈ Follow(T)	reduce T→int

Parsing SLR

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

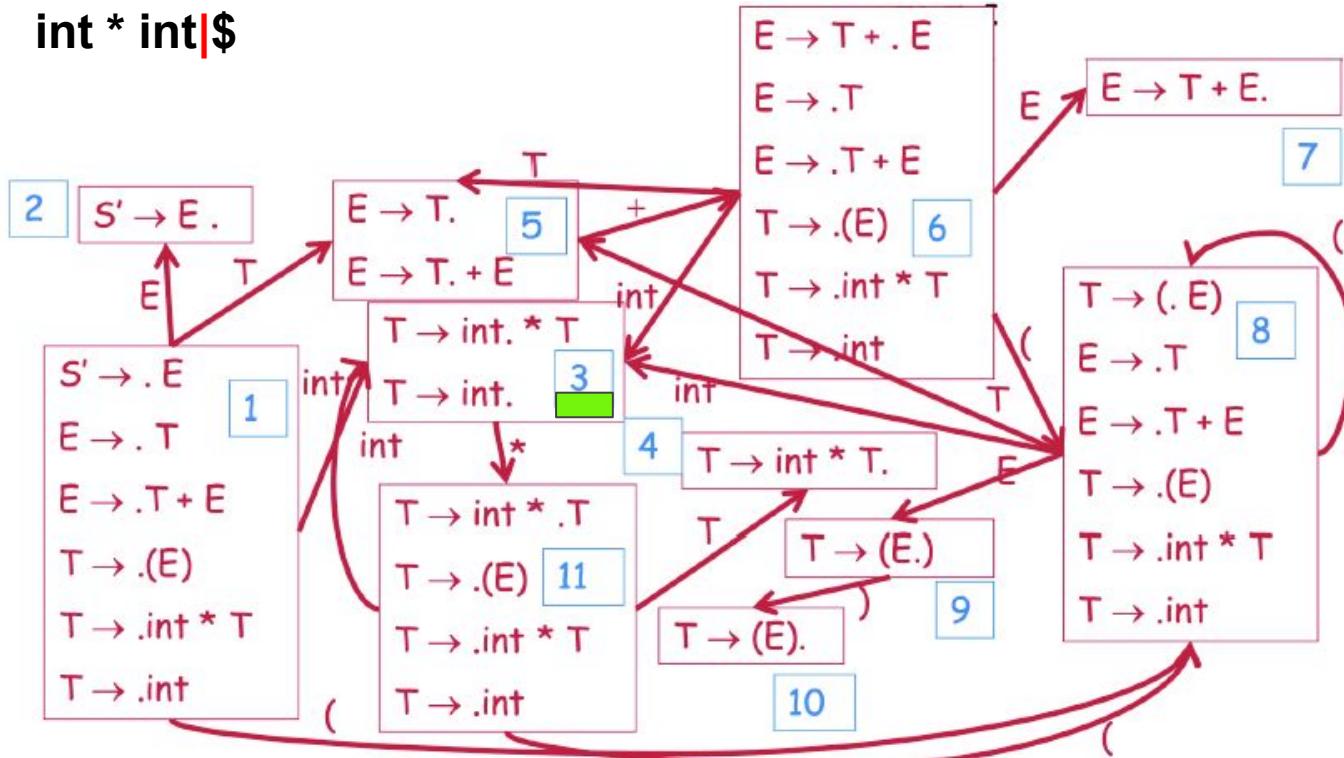
int * int|\$



Parsing SLR

$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} \mid (E) \end{aligned}$$

int * int|\$

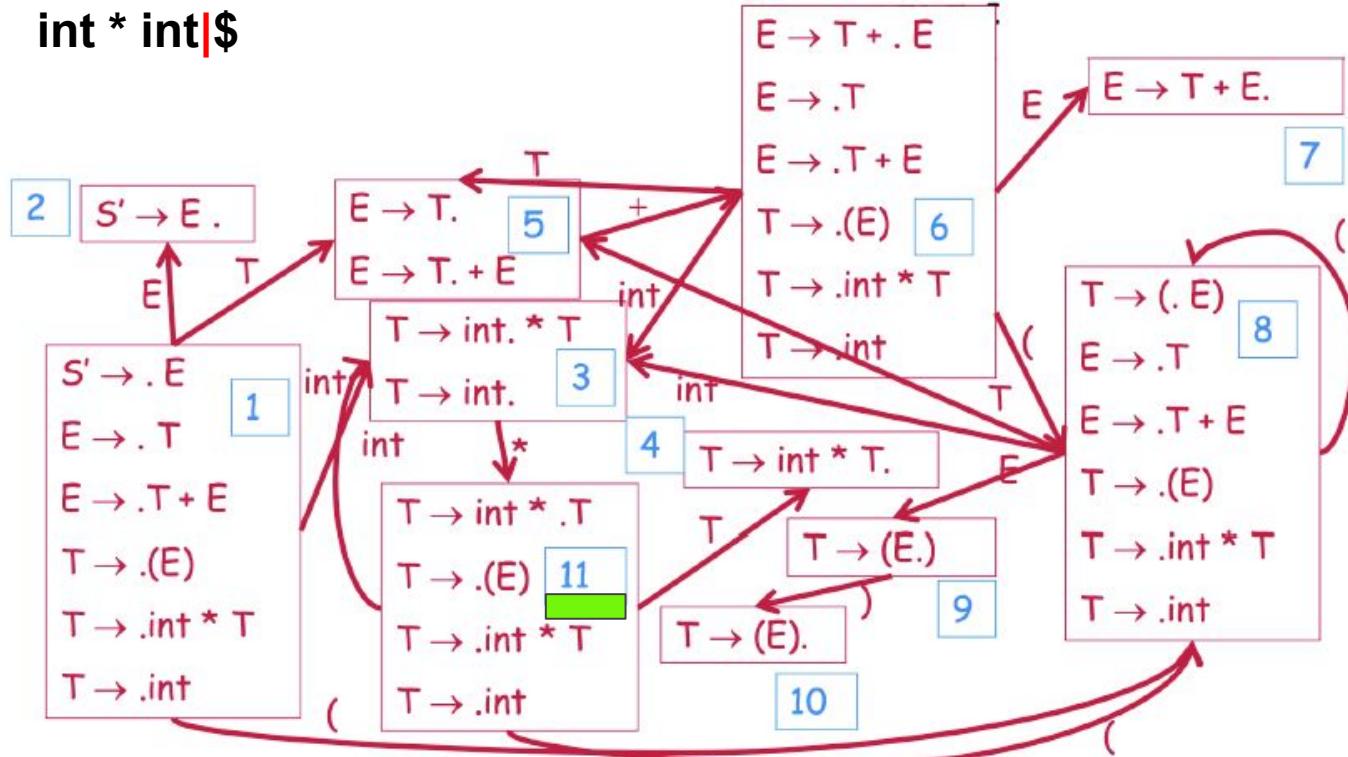


Parsing SLR

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

int * int|\$

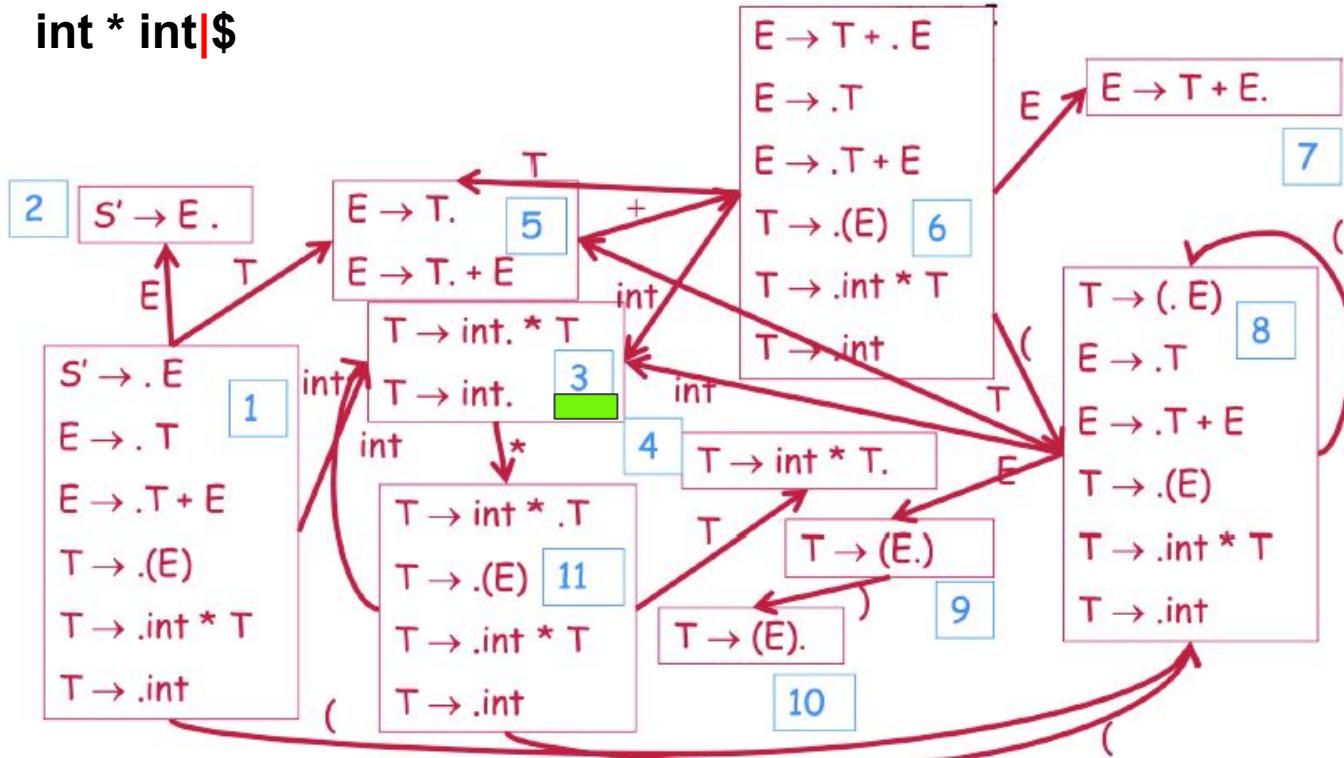


Parsing SLR

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

int * int|\$





Parsing SLR

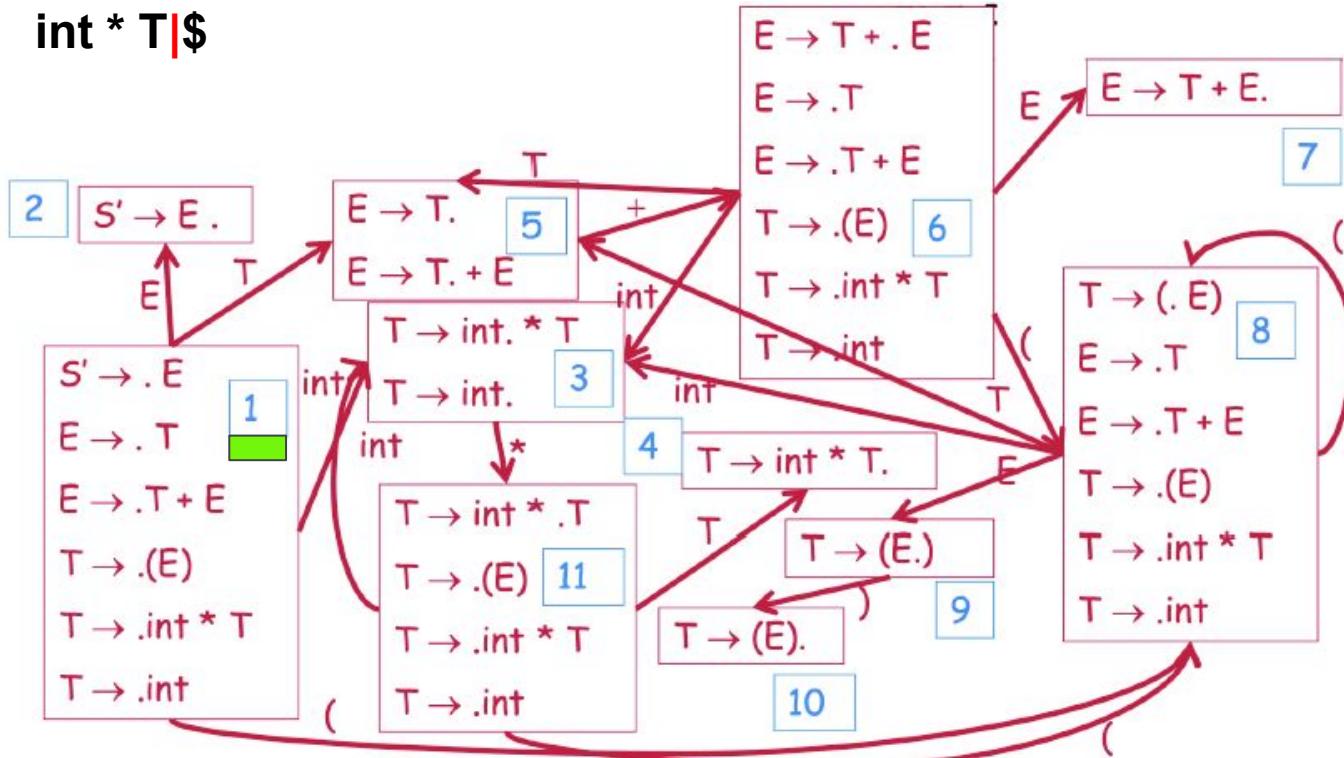
$\text{Follow}(E) = \{ ')', \$ \}$
 $\text{Follow}(T) = \{ '+', ')', \$ \}$

Entrada	Estado DFA	Acción
int * int\$	1	shift
int * int\$	3 * no es follow de T	shift
int * int\$	11	shift
int * int \$	3 \$ \in Follow(T)	reduce T \rightarrow int
int * T \$	4 \$ \in Follow(T)	reduce T \rightarrow int*T

Parsing SLR

$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} \mid (E) \end{aligned}$$

int * T | \$

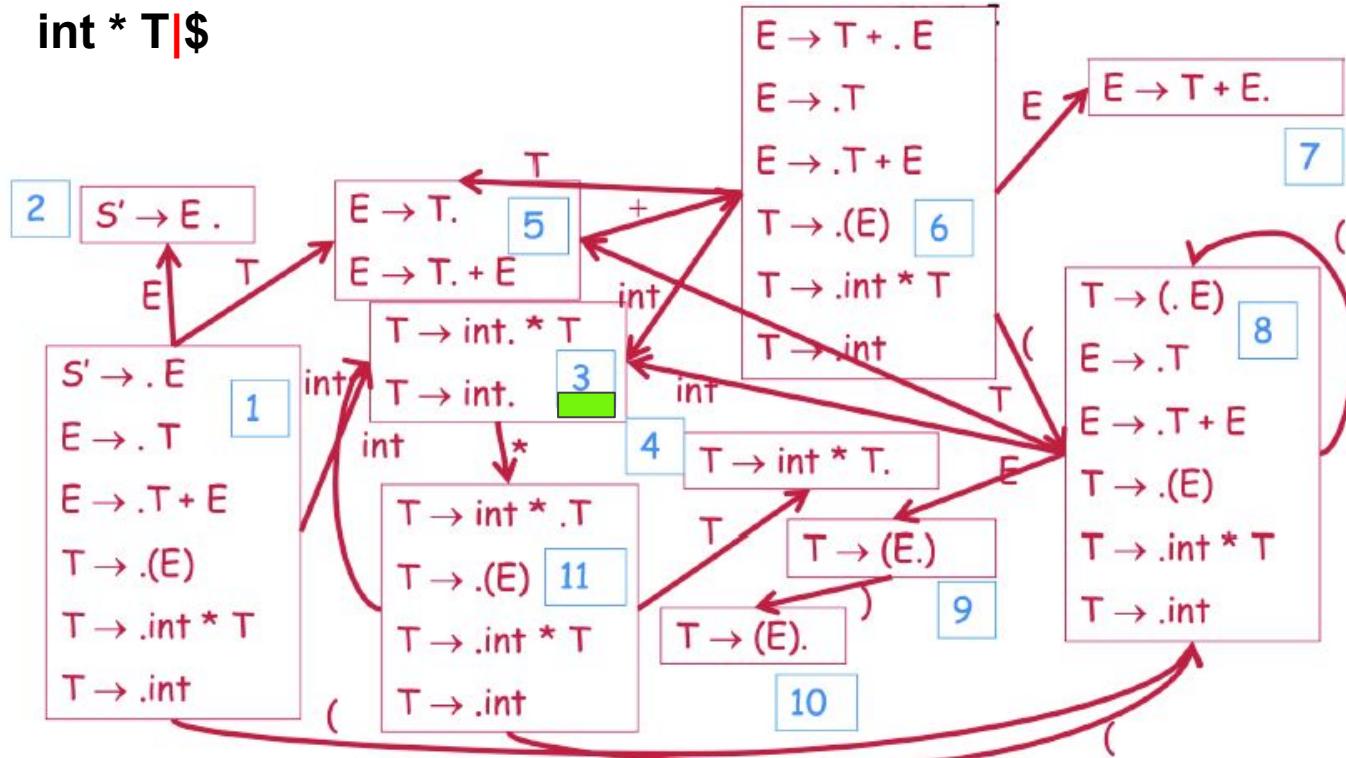


Parsing SLR

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

int * T | \$

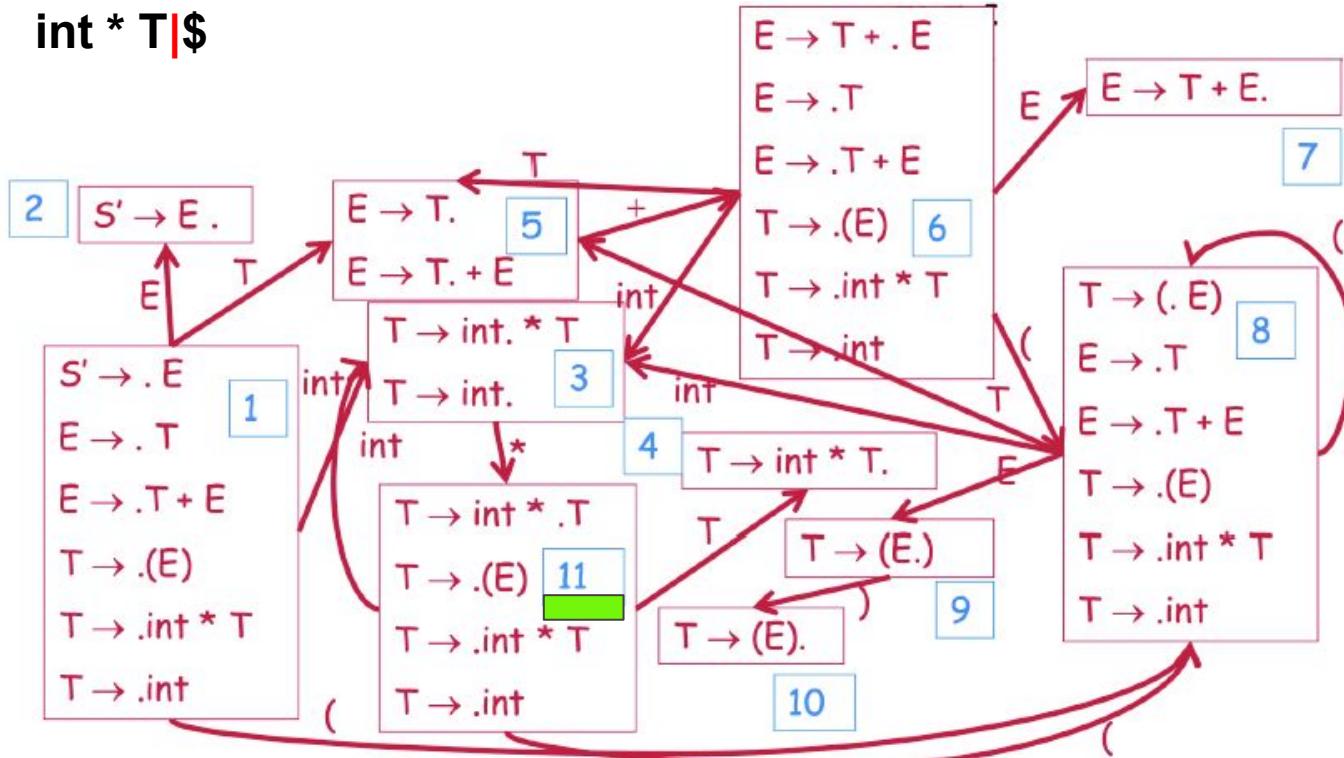


Parsing SLR

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

int * T | \$

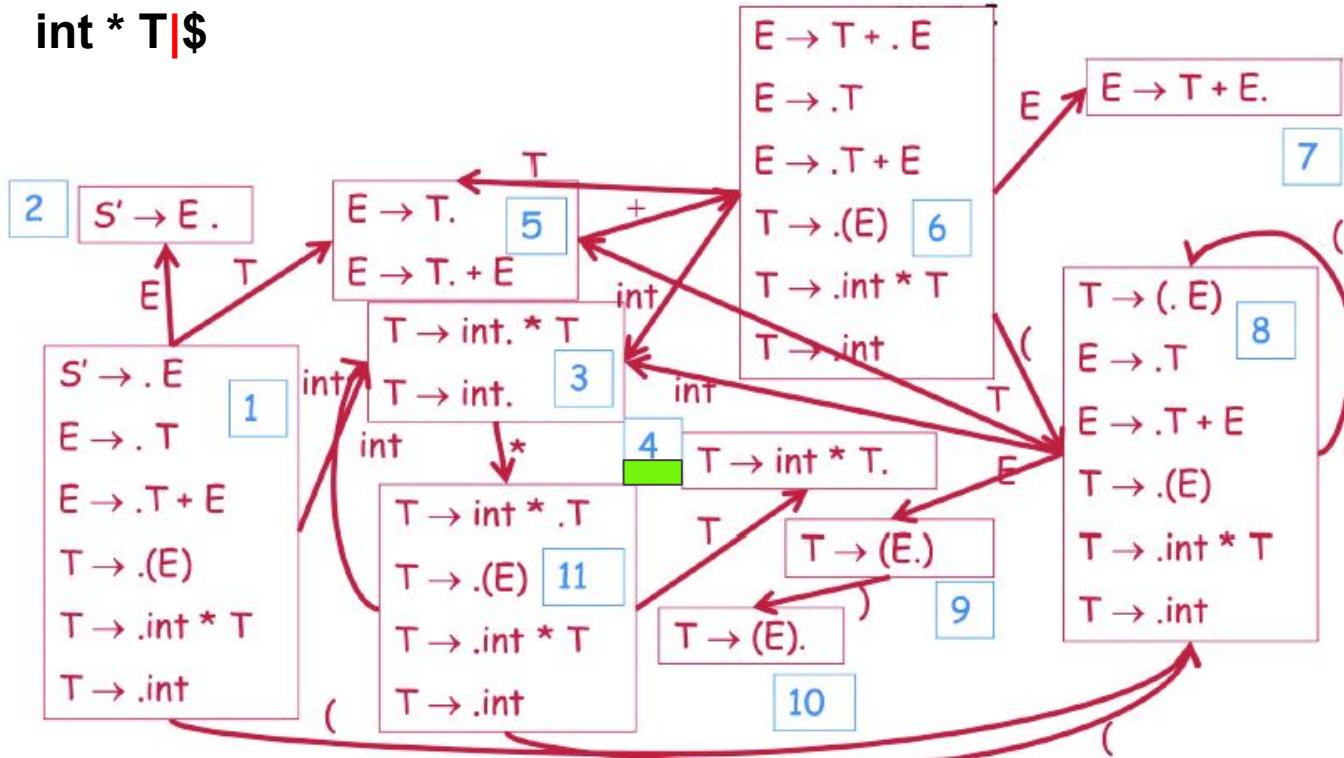


Parsing SLR

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

int * T | \$





Parsing SLR

$$\begin{aligned}\text{Follow}(E) &= \{ ')', \$ \} \\ \text{Follow}(T) &= \{ '+', ')', \$ \}\end{aligned}$$

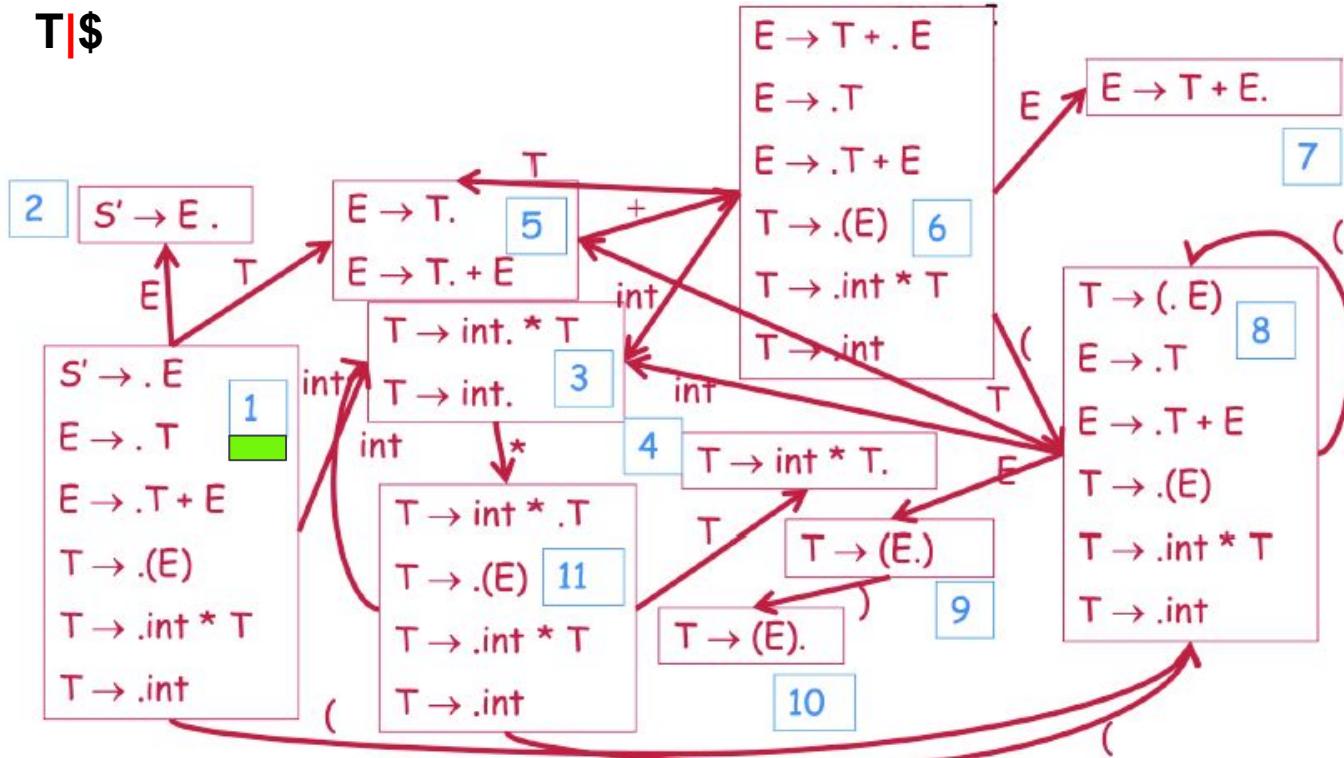
Entrada	Estado DFA	Acción
int * int\$	1	shift
int * int\$	3 * no es follow de T	shift
int * int\$	11	shift
int * int \$	3 \$ \in Follow(T)	reduce T \rightarrow int
int * T \$	4 \$ \in Follow(T)	reduce T \rightarrow int*T
T \$	5 \$ \in Follow(T)	reduce E \rightarrow T

Parsing SLR

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

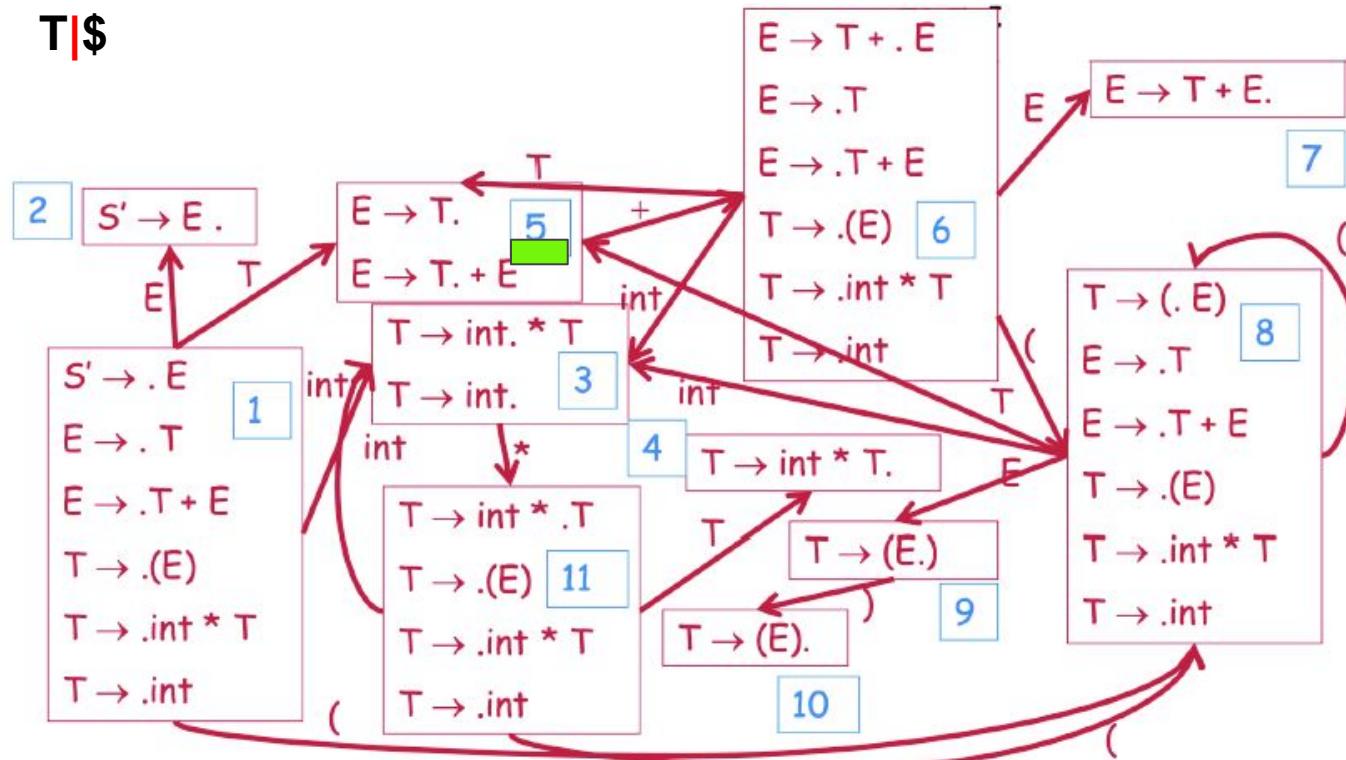
$T \mid \$$



Parsing SLR

$E \rightarrow T + E \mid T$
 $T \rightarrow \text{int} * T \mid \text{int} \mid (E)$

T|\$





Parsing SLR

$\text{Follow}(E) = \{ ')', \$ \}$
 $\text{Follow}(T) = \{ '+', ')', \$ \}$

Entrada	Estado DFA	Acción
int * int\$	1	shift
int * int\$	3 * no es follow de T	shift
int * int\$	11	shift
int * int \$	3 \$ \in Follow(T)	reduce T \rightarrow int
int * T \$	4 \$ \in Follow(T)	reduce T \rightarrow int*T
T \$	5 \$ \in Follow(T)	reduce E \rightarrow T
E \$		accept



Parsing SLR

Una mejora

- Ejecutar el autómata en cada paso es ineficiente.
 - La mayor parte del trabajo se repite.
- Recordar el estado del autómata en cada prefijo del stack.
- Cambiar el stack para que contenga pares
⟨ símbolo, estado del DFA ⟩



Parsing SLR

Una mejora

- Para un stack
 $\langle \text{símbolo}_1, \text{estado}_1 \rangle \dots \langle \text{símbolo}_n, \text{estado}_n \rangle$,
el **estado_n** es el estado final del DFA sobre **símbolo₁...símbolo_n**.
- Detalle: La parte inferior del stack es $\langle \text{dummy}, \text{start} \rangle$, donde
 - **dummy** es un símbolo ficticio
 - **start** es el estado inicial del DFA.



Parsing SLR

Goto (DFA) Table

Definir $\text{goto}[i,A] = j$ si $\text{estado}_i \xrightarrow{A} \text{estado}_j$

goto es simplemente la función de transición del DFA.

- Una de las dos tablas de análisis.



Parsing SLR

Refinando los movimientos del Parser

- Shift x
 - Empujar $\langle a, x \rangle$ en la pila
 - a es la entrada actual
 - x es un estado del DFA
- Reduce $X \rightarrow \alpha$
 - Como anteriormente
- Accept
- Error



Parsing SLR

Action Table

Para cada estado s_i and terminal t

- Si s_i tiene el item $X \rightarrow \alpha.t\beta$ and $\text{goto}[i,t] = k$ entonces $\text{action}[i,t] = \text{shift } k$
- Si s_i tiene el item $X \rightarrow \alpha.$ and $t \in \text{Follow}(X)$ y $X \neq S'$ entonces
 $\text{action}[i,t] = \text{reduce } X \rightarrow \alpha$
- Si s_i tiene el item $S' \rightarrow S.$ then $\text{action}[i,\$] = \text{accept}$
- De lo contrario, $\text{action}[i,t] = \text{error}$



Algoritmo SLR

Let input = w\$ be initial input

Let j = 0

Let DFA state 1 be the one with item $S' \rightarrow .S$

Let stack = ⟨ dummy, 1 ⟩ // ⟨ symbol, state ⟩

repeat

 case action[top_state(stack), input[j]] of

 shift k: push ⟨ input[j++], k ⟩

 reduce $X \rightarrow \alpha$:

 pop $|\alpha|$ pairs,

 push ⟨ X , goto[top_state(stack), X] ⟩

 accept: halt normally

 error: halt and report error



Parsing SLR

Tomar en cuenta que el algoritmo utiliza solo los estados del DFA y la entrada.

¡Los símbolos del stack nunca se utilizan!

Sin embargo, aún se necesitan los símbolos para las acciones semánticas.



Parsing SLR

- Algunas construcciones comunes no son SLR(1).
- LR(1) es más poderoso.
 - Incorpora la anticipación en los ítems.
 - Un ítem LR(1) es un par: (**ítem LR(0)**, **x lookahead**).
 - **[T → . int * T, \$]** significa
 - Despues de ver **T → int * T**, **reduce si el lookahead es \$**.
 - Más preciso que simplemente usar conjuntos de follow.