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”

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

- 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.

producciones

int * **int** + int
int * **T** + int
T + int
T + **T**
T + **E**
E

reducciones

símbolo inicial

```
T → int
T → int * T
T → int
E → T
E → T + E
```

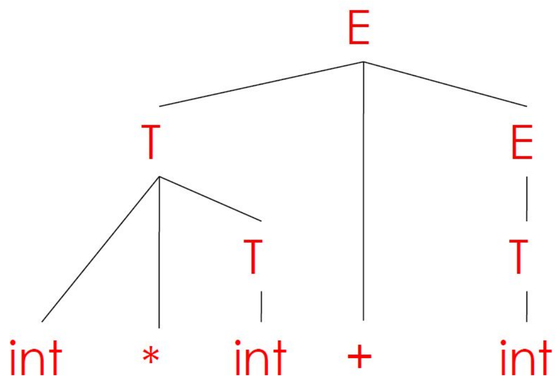
PARSERS BOTTOM-UP

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

T

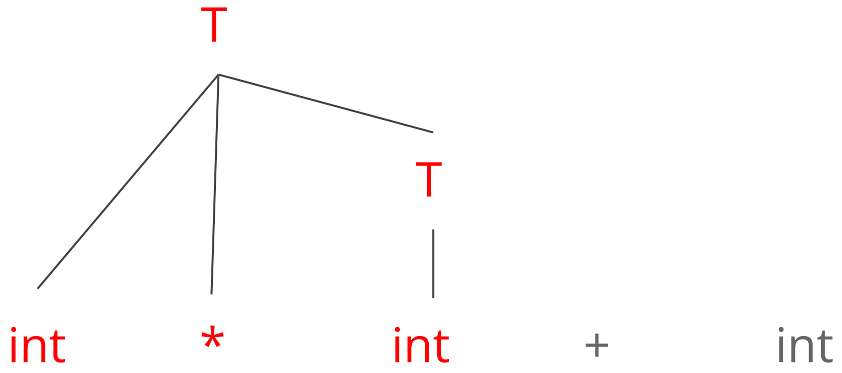
int * int + int

PARSERS BOTTOM-UP

int * int + int

int * T + int

T + int



PARSERS BOTTOM-UP

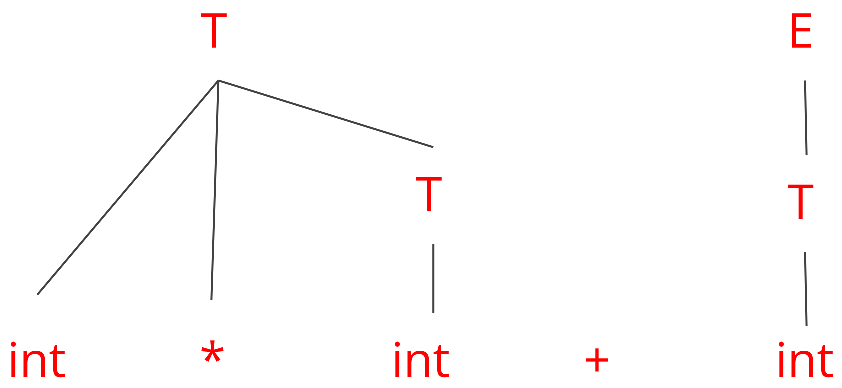
int * int + int

int * T + int

T + int

T + T

T + E



PARSERS BOTTOM-UP

int * int + int

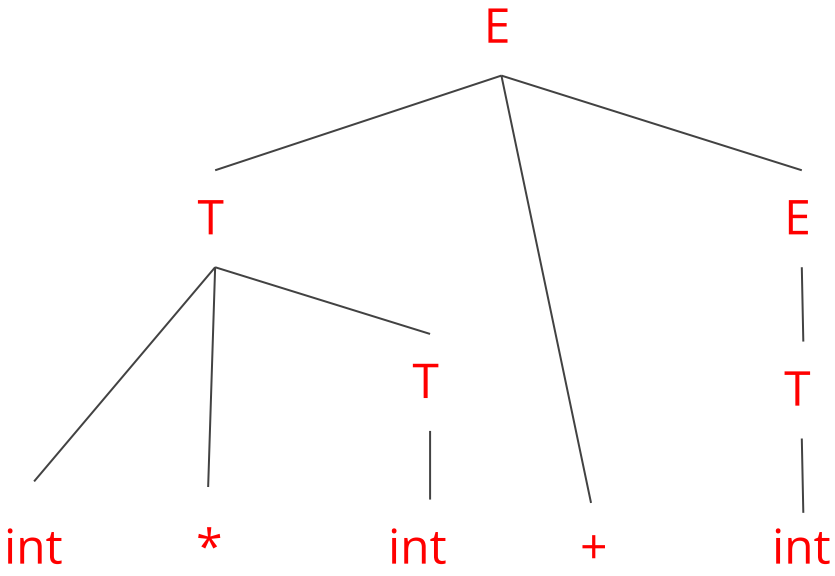
int * T + int

T + int

T + T

T + E

E



PARSERS BOTTOM-UP

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

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

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

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 . . . xn

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 $\rightarrow$ int
int * T |+ int	reduce T $\rightarrow$ int * T
T |+ int	shift
T + |int	shift
T + int|	reduce T $\rightarrow$ 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

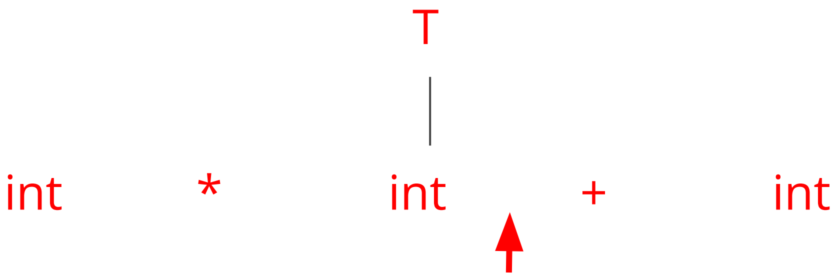
|int * int + int

int |* int + int

int * |int + int

int * int |+ int

int * T |+ int



SHIFT-REDUCE

|int * int + int

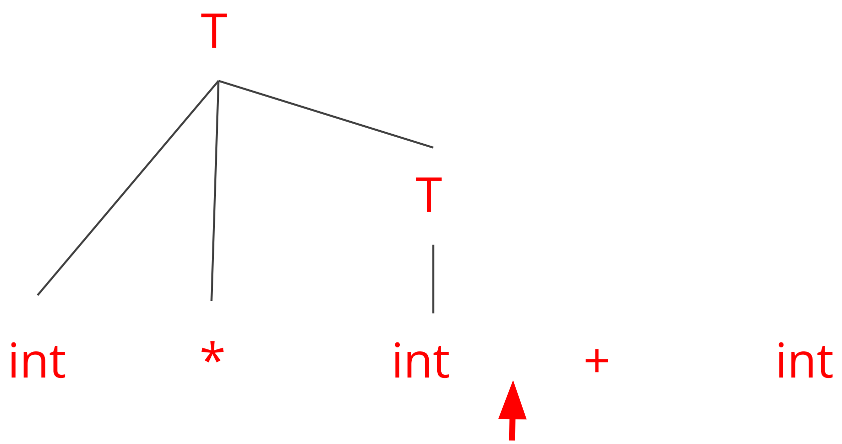
int |* int + int

int * |int + int

int * int |+ int

int * T |+ int

T |+ int



SHIFT-REDUCE

|int * int + int

int |* int + int

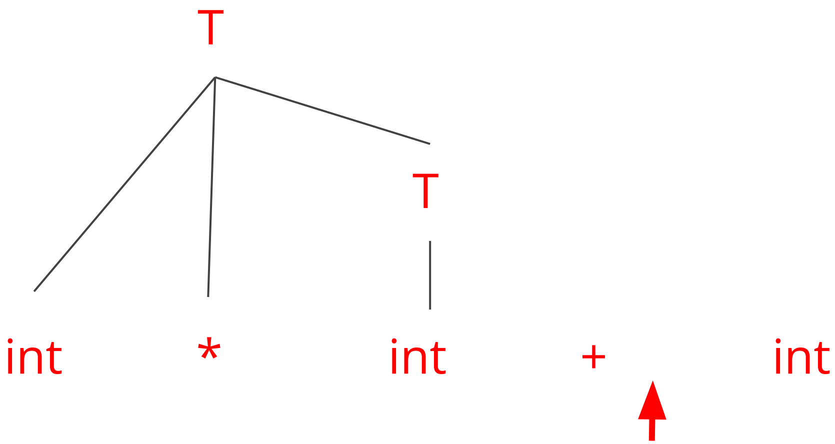
int * |int + int

int * int |+ int

int * T |+ int

T |+ int

T + |int



SHIFT-REDUCE

|int * int + int

int |* int + int

int * |int + int

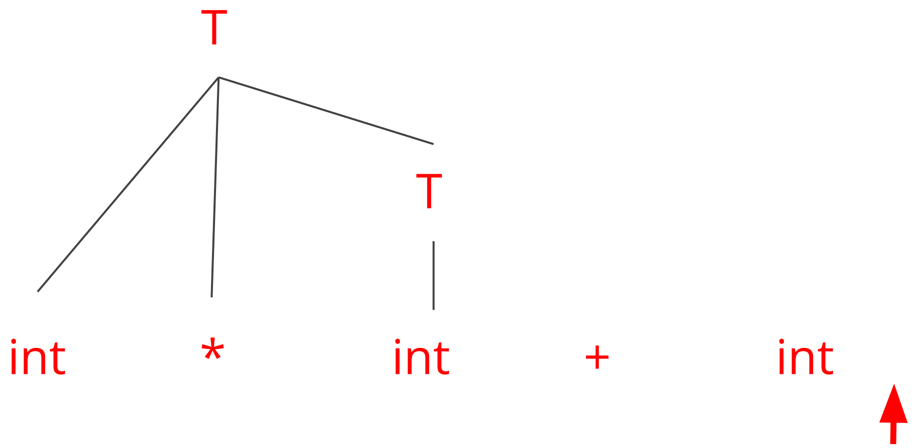
int * int |+ int

int * T |+ int

T |+ int

T + |int

T + int|



SHIFT-REDUCE

|int * int + int

int |* int + int

int * |int + int

int * int |+ int

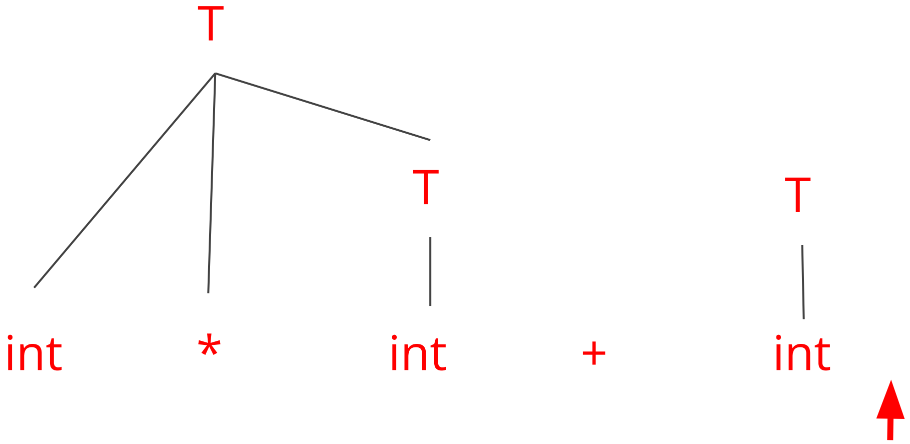
int * T |+ int

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

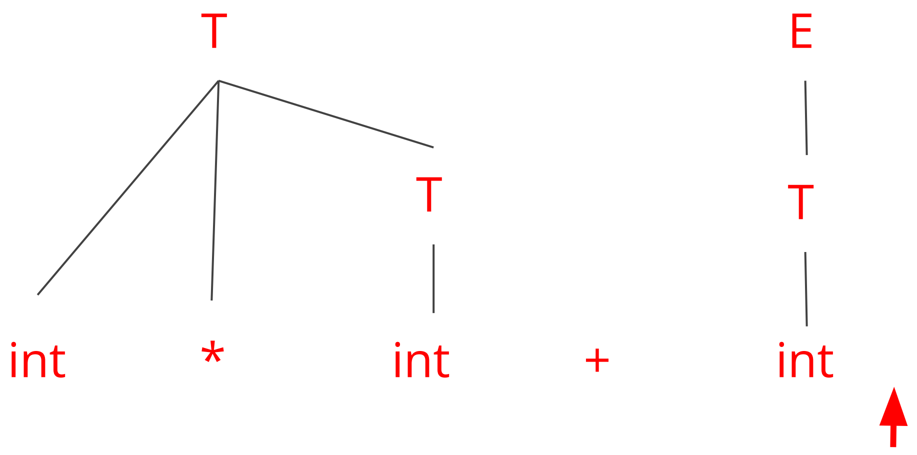
T |+ int

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

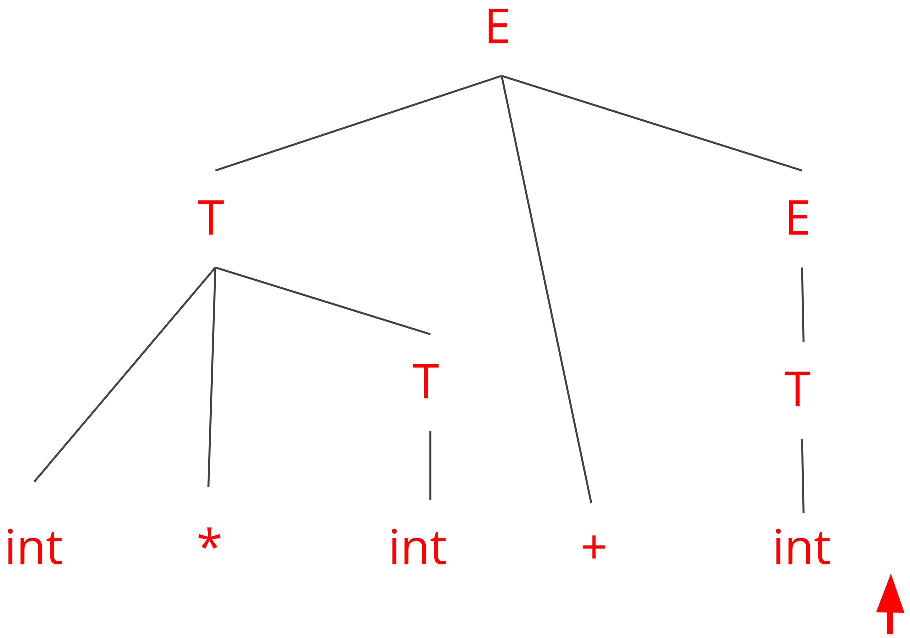
T + |int

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:

```
E → T + E|T
T → int|int * T|(E)
```

- Considere paso **int| * int + int**
- Podríamos reducir por **T → int** mediante **T| * int + 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 **αβ** es un handle de **αβω**

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

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

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

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

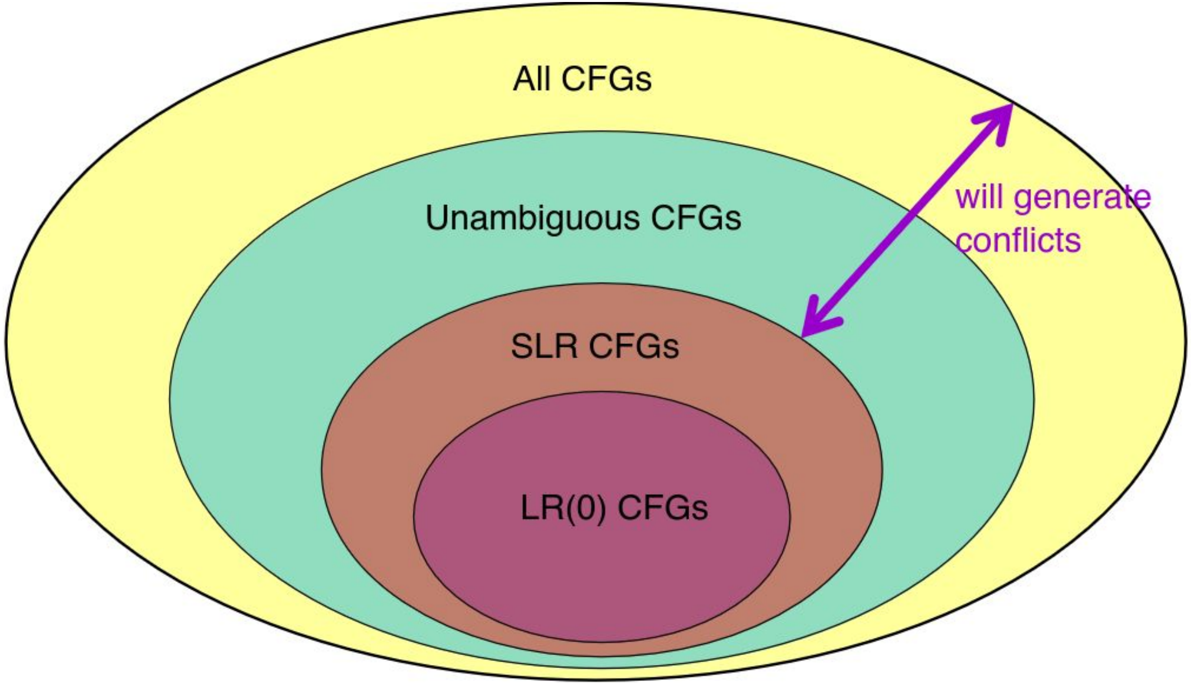
Trabajando con Handles

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 **α|ω** 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 → (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:
$E \rightarrow T + E | T$
$T \rightarrow int * T | int | (E)$

- 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:
  $\boldsymbol{Prefix_1\ Prefix_2 \ldots Prefix_{n-1}\ Prefix_n}$
- Sea $\boldsymbol{Prefix_i}$ un prefijo el lado derecho de la producción $\boldsymbol{X_i} \rightarrow \boldsymbol{\alpha_i}$
  - $\boldsymbol{Prefix_i}$ eventualmente se reducirá a $\boldsymbol{X_i}$
  - La parte que falta de $\boldsymbol{\alpha_{i-1}}$ comienza con $\boldsymbol{X_i}$
  - Es decir, hay un $\boldsymbol{X_{i-1}} \rightarrow \boldsymbol{Prefix_{i-1} X_i \beta}$ para algún $\boldsymbol{\beta}$
- De forma recursiva, $\boldsymbol{Prefix_{k+1} \ldots Prefix_n}$ finalmente se reduce a la parte que falta de $\boldsymbol{\alpha_k}$

Trabajando con Handles

La “pila de ítems”

$T \rightarrow (.E)$

$E \rightarrow .T$

$T \rightarrow int * .T$

Significa:

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

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 Viables

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 $E \rightarrow \alpha.X\beta \rightarrow_{X} E \rightarrow \alpha X.\beta$
4. Para el ítem $E \rightarrow \alpha.X\beta$ y la producción $X \rightarrow \Delta$, agregue
5. $E \rightarrow \alpha.X\beta \rightarrow_{\varepsilon} X \rightarrow .\Delta$
6. Cualquier estado es un **estado de aceptación**
7. El estado inicial es $S' \rightarrow .S$

Reconociendo Prefixes Viables

$S \longrightarrow E$
$E \longrightarrow T + E | T$
$T \longrightarrow int * T | int | (E)$

Definimos el NFA...

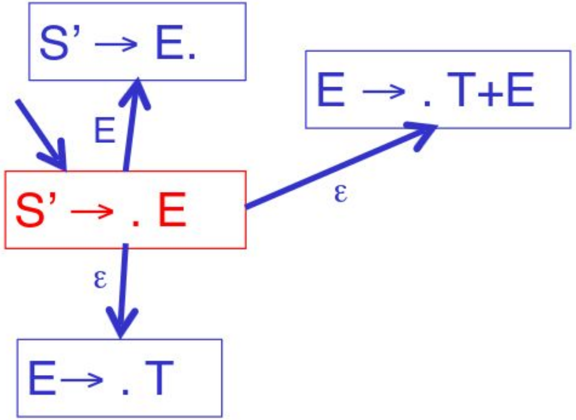
Reconociendo Prefixes Viables

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

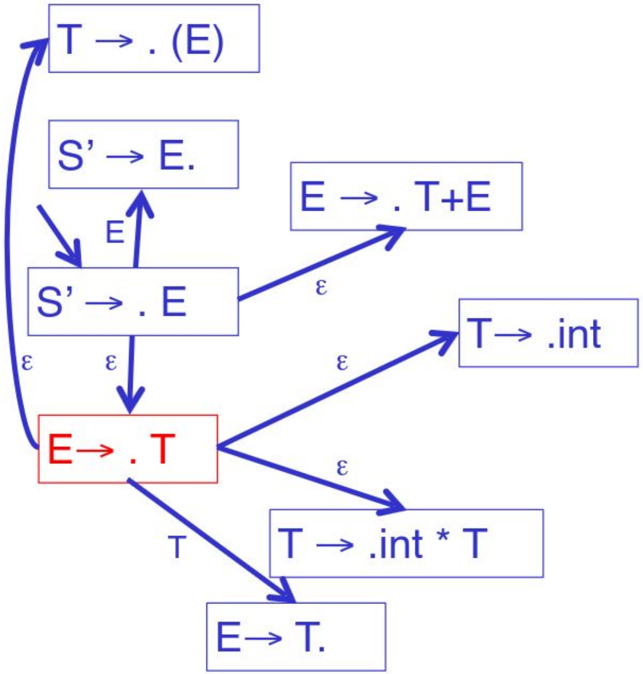
$\boxed{S' \rightarrow . E}$

Reconociendo Prefixes Viables

E → T + E | T
T → int * T | int | (E)

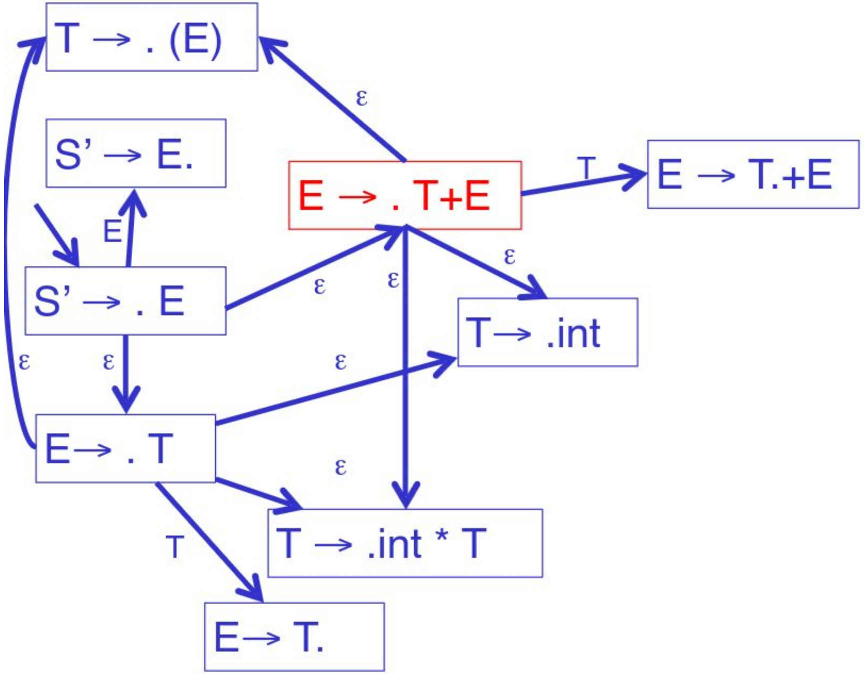


Reconociendo Prefixes Viables



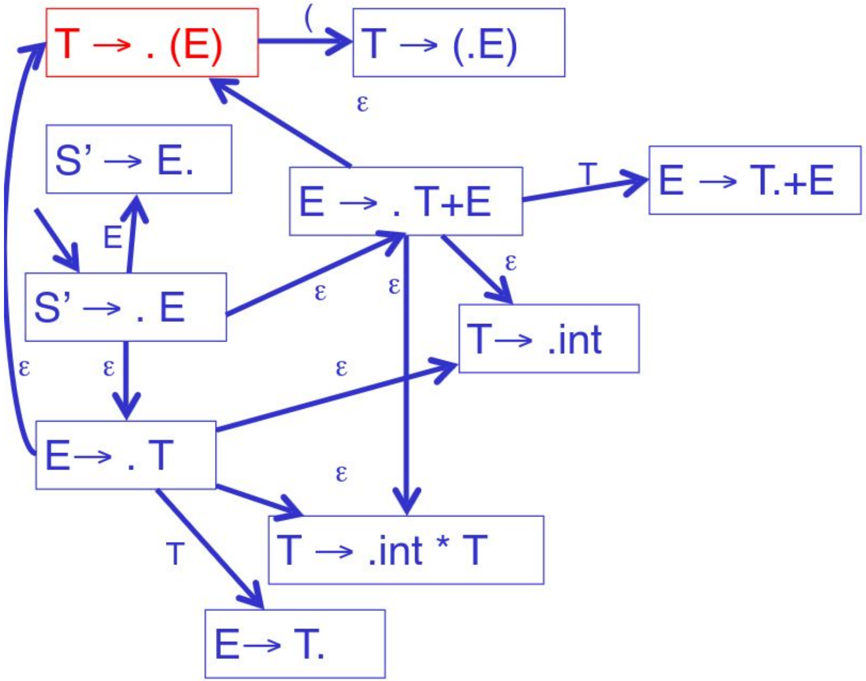
E → T + E | T
T → int * T | int | (E)

Reconociendo Prefixes Viables



E → T + E | T
T → int * T | int | (E)

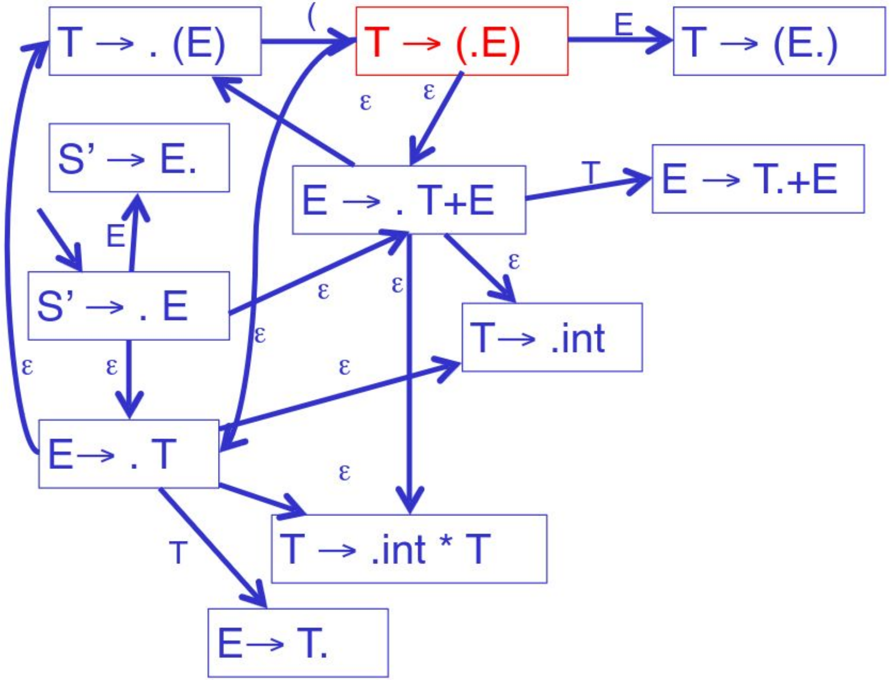
Reconociendo Prefixes Viables



$E \rightarrow T + E \mid T$

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

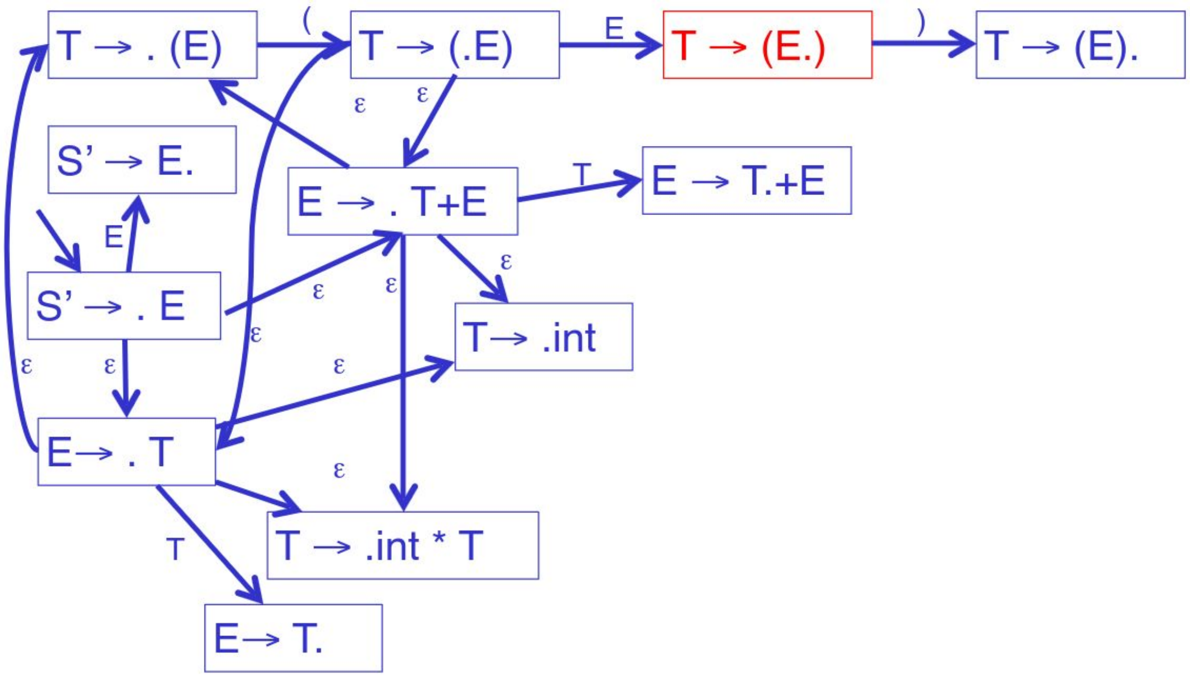
Reconociendo Prefixes Viables



E → T + E | T

T → int * T | int | (E)

Reconociendo Prefixes Viables

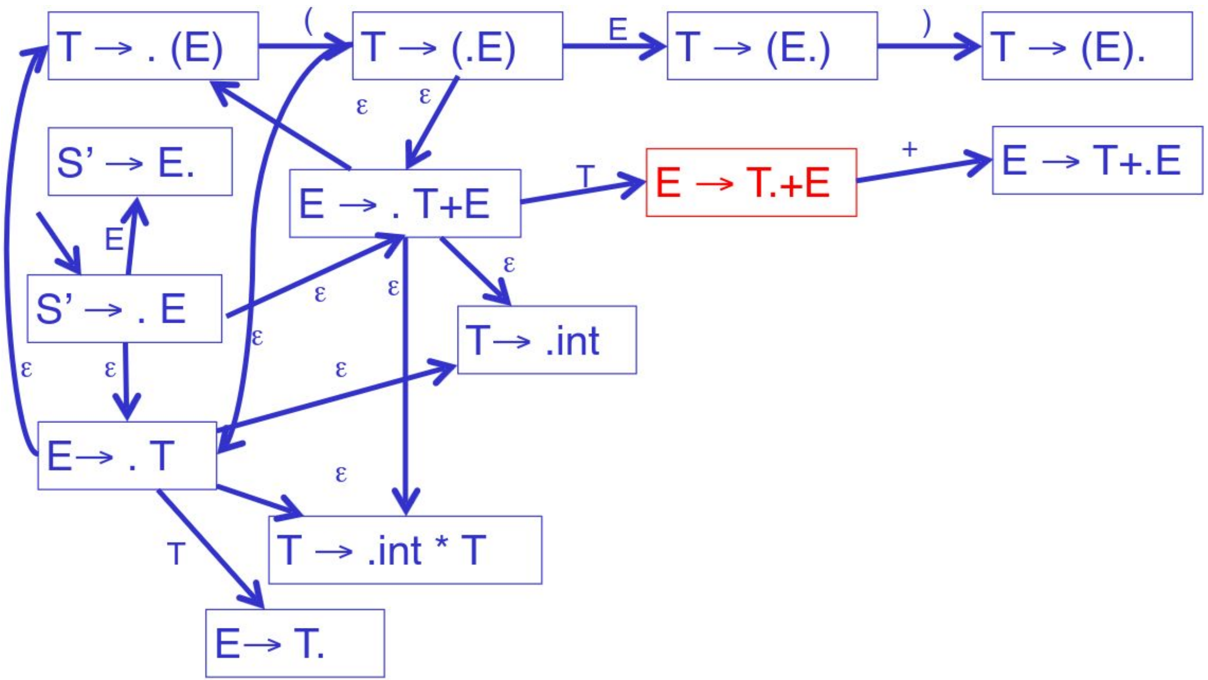


E → T + E | T

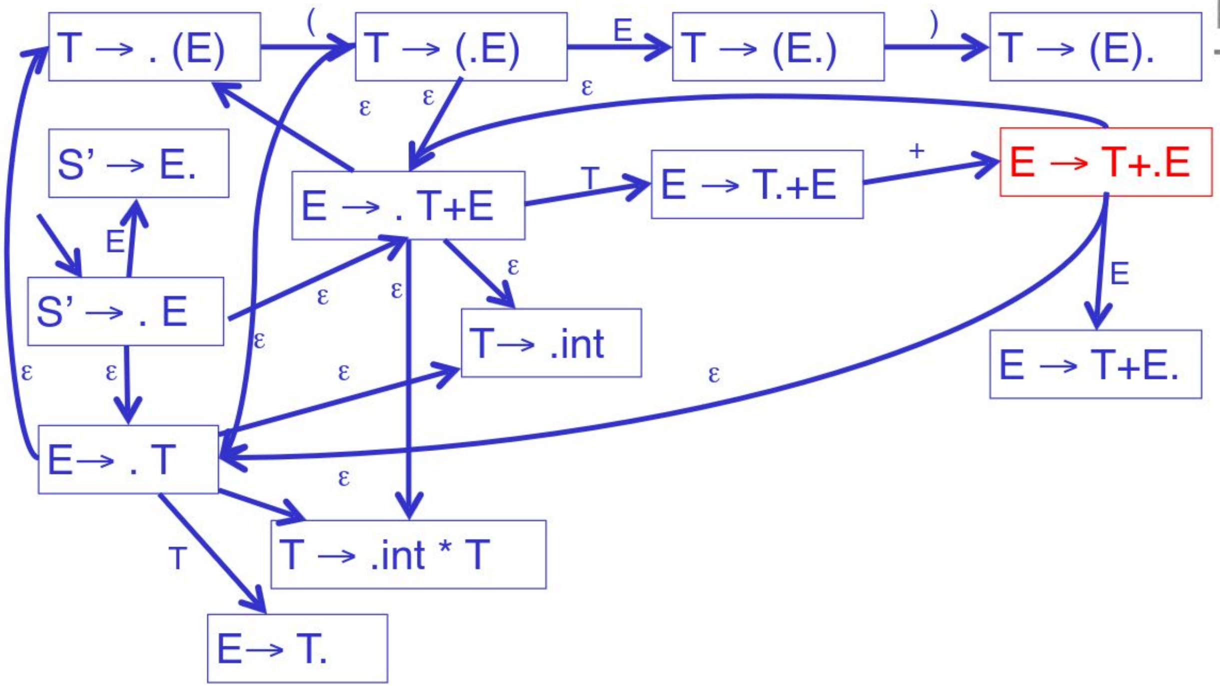
T → int * T | int | (E)

Reconociendo Prefixes Viables

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



Reconociendo Prefixes Viables



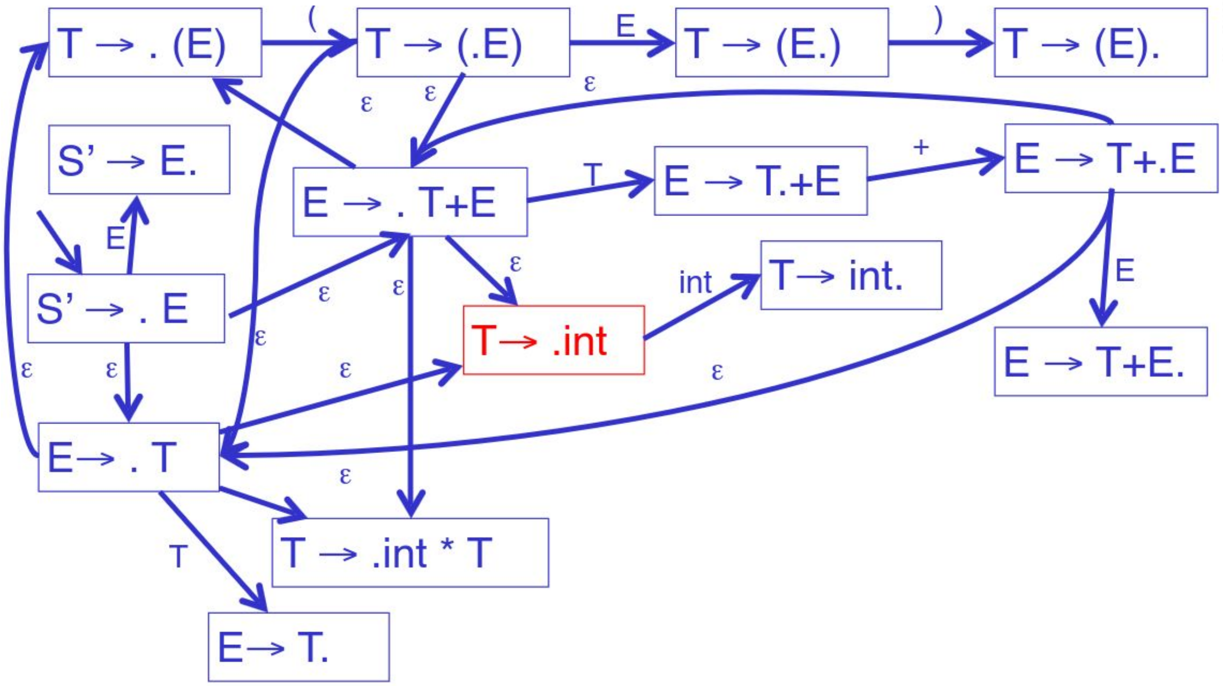
$E \rightarrow T + E \mid T$

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

Reconociendo Prefixes Viables

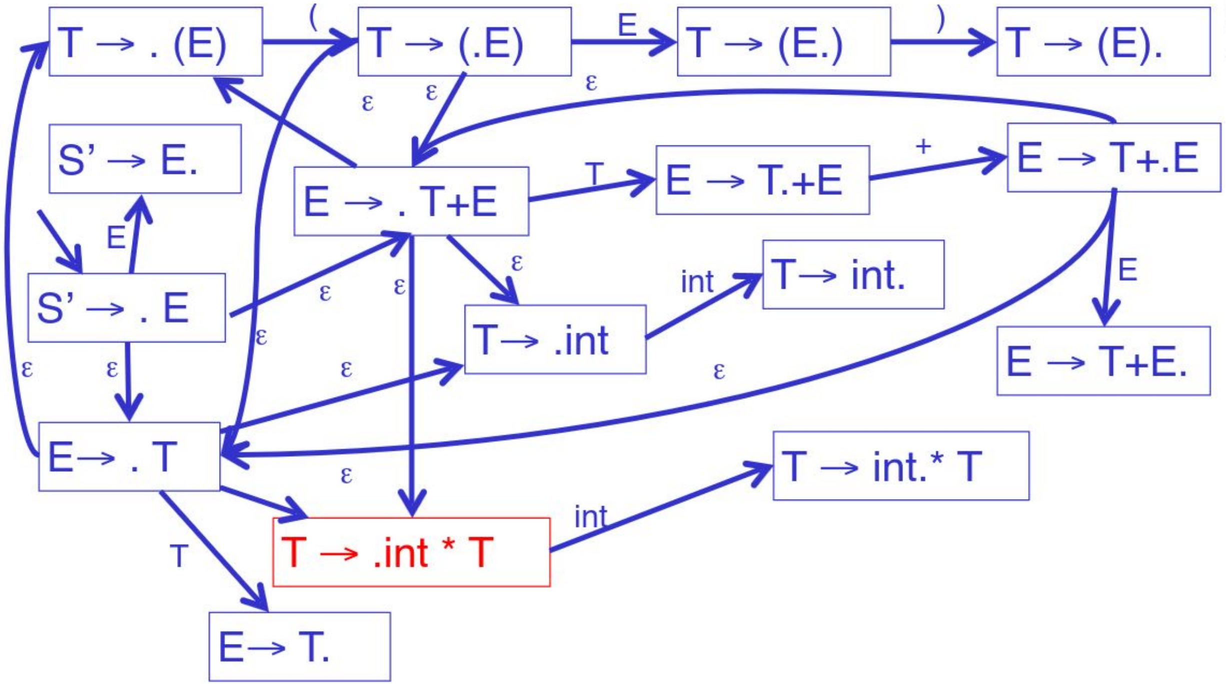
$E \rightarrow T + E \mid T$

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

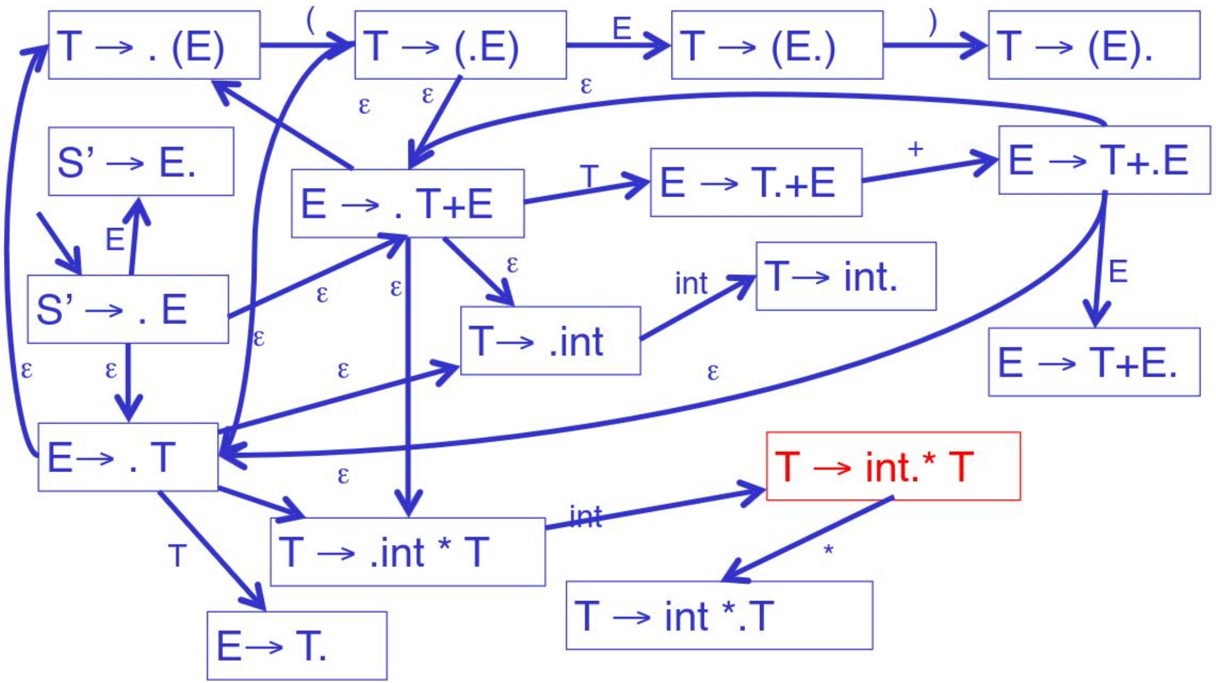


Reconociendo Prefixes Viables

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

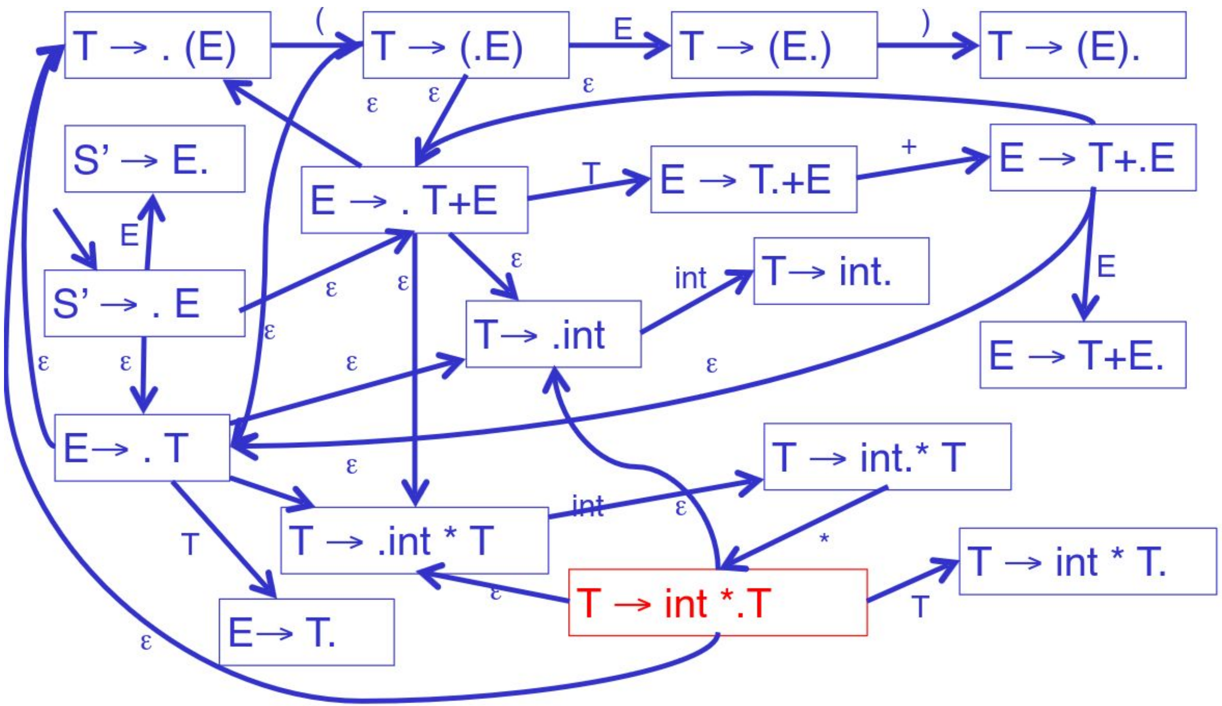


Reconociendo Prefixes Viables



E → T + E | T
T → int * T | int | (E)

Reconociendo Prefixes Viables

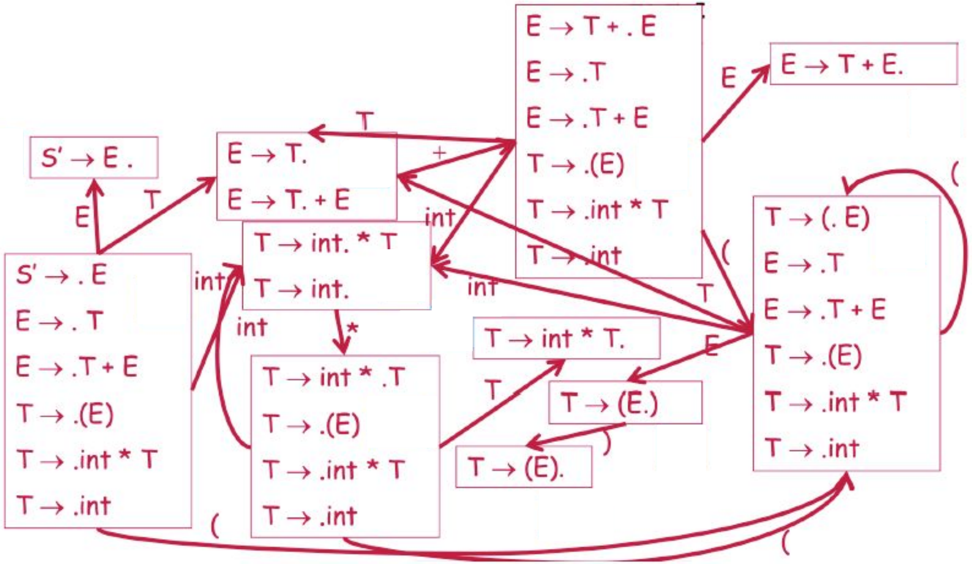


$E \rightarrow T + E \mid T$

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

Reconociendo Prefixes Viables

Convirtiendo en DFA



Reconociendo Prefixes Viables

Item Válidos

El Ítem $\boldsymbol{X \rightarrow \beta.\gamma}$ es válido para un prefijo viable $\boldsymbol{\alpha\beta}$ si

$$S' \rightarrow^* \alpha X \omega \rightarrow \alpha\beta\gamma\omega$$

para una derivación por la derecha.

Después de analizar $\boldsymbol{\alpha\beta}$, 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*** → ***(.E)*** es válido para los prefijos:

(

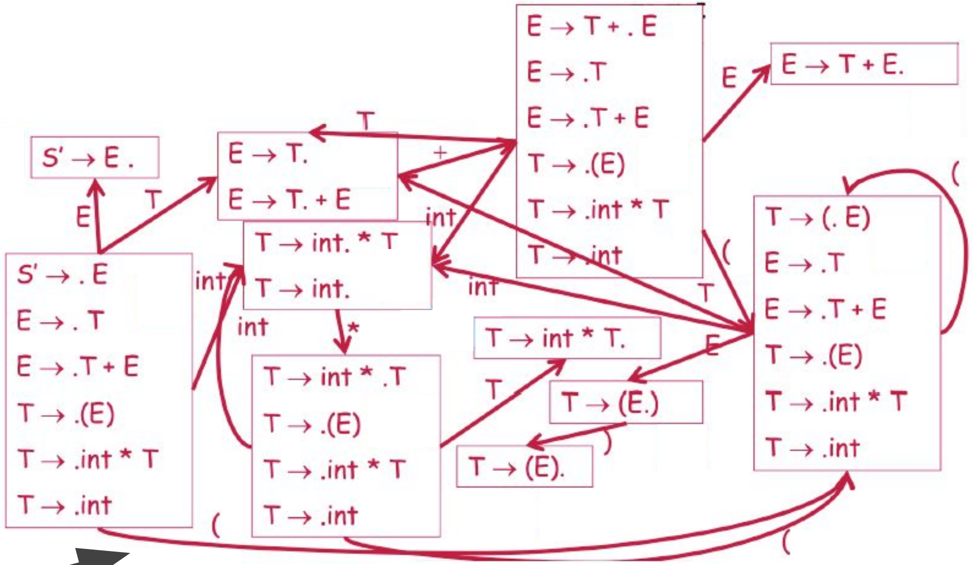
((

(((

((((

. . .

Reconociendo Prefixes Viables



continuamos iterando para ***((((..***

Cadena de entrada: ***((((..***

Parsing LR(0) y SLR

Parsing LR(0)

Idea: Supongamos que

– el stack contiene $\boldsymbol{\alpha}$

– el siguiente símbolo de entrada es $\boldsymbol{t}$

– el DFA con la entrada $\boldsymbol{\alpha}$ termina en el estado $\boldsymbol{s}$

- Reducir por $\boldsymbol{X} \to \boldsymbol{\beta}$ si
  – $\boldsymbol{s}$ contiene el ítem $\boldsymbol{X} \to \boldsymbol{\beta}.$
- Hacer shift sí
  – $\boldsymbol{s}$ contiene el ítem $\boldsymbol{X} \to \boldsymbol{\beta} \cdot \boldsymbol{t}\ \boldsymbol{\omega}$
  – es equivalente a decir que $\boldsymbol{s}$ tiene una transición etiquetada con $\boldsymbol{t}$.

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 $Y \rightarrow \omega \, . \, t \, \delta$

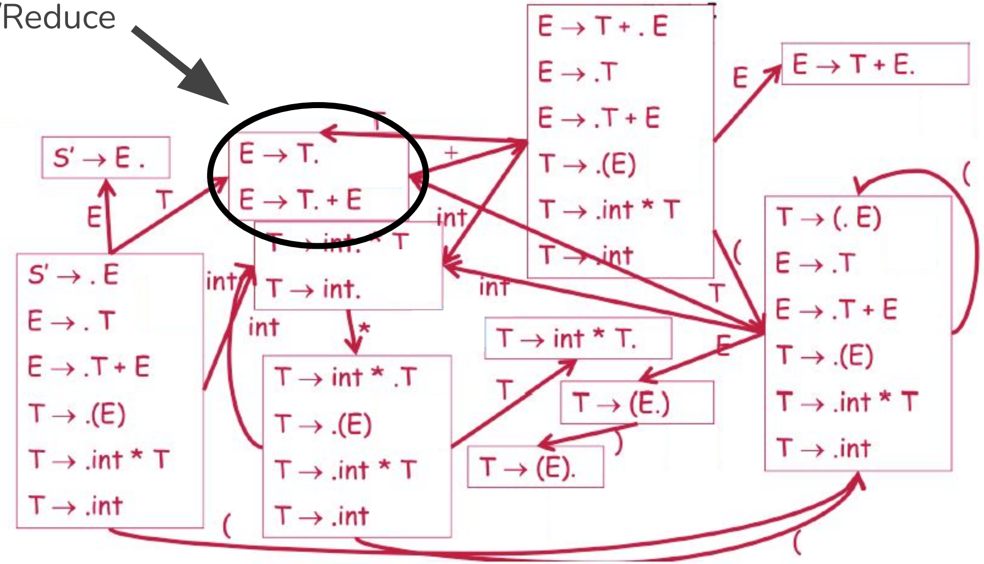
Parsing LR(0)

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**.

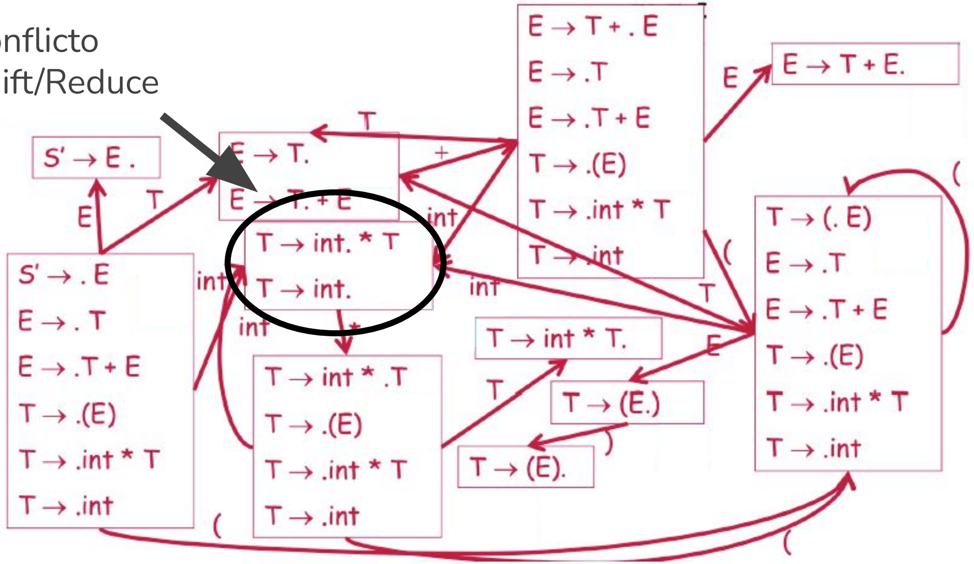
Reconociendo Prefixes Viables

Conflicto
Shift/Reduce



Reconociendo Prefixes Viables

Conflicto
Shift/Reduce



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 $\boldsymbol{\alpha}$

– el siguiente símbolo de entrada es $\boldsymbol{t}$

– el DFA con la entrada $\boldsymbol{\alpha}$ termina en el estado $\boldsymbol{s}$

- Reducir por $\boldsymbol{X \rightarrow \beta}$ si
  – $\boldsymbol{s}$ contiene el ítem $\boldsymbol{X \rightarrow \beta.}$
  – $\boldsymbol{t \in Follow(X)}$
- Hacer shift sí
  – $\boldsymbol{s}$ contiene el ítem $\boldsymbol{X \rightarrow \beta \cdot t\ \omega}$

Parsing SLR

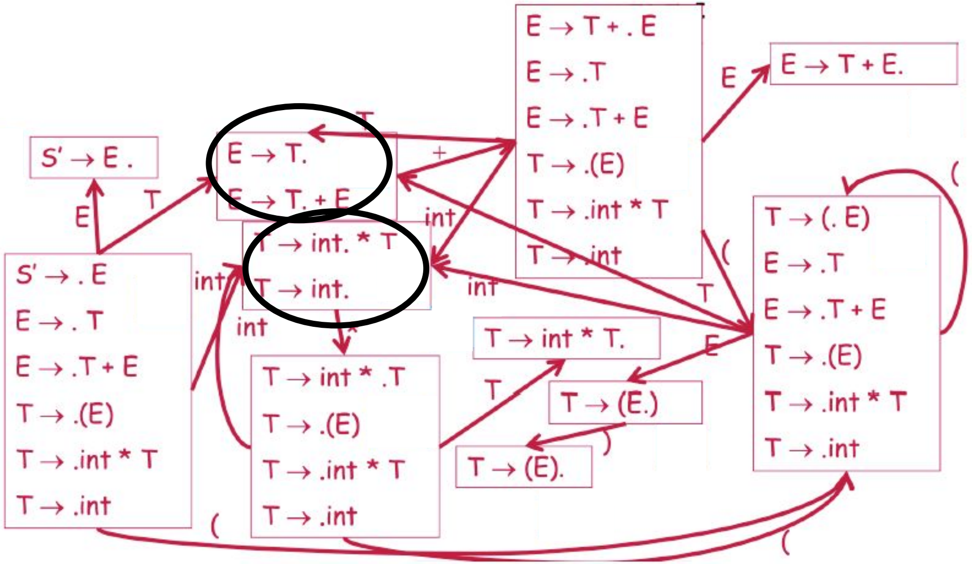
Si hay conflictos bajo estas reglas, la gramática no es SLR.

Las reglas equivalen a una heurística para detectar handles

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

Parsing SLR

Follow(E) = { ')', $ }
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:

$$E \to E + E \mid E * E \mid (E) \mid int$$

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

- $E \to E * E\,.$
- $E \to E\,.\,+ E$

¡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 **$|x_1...x_n\$$** 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 $\mathbf{X \to \beta.} \in \mathbf{L}$ y $\mathbf{t} \in \mathbf{Follow(X)}$
  - De lo contrario, shift si $\mathbf{X \to \beta.\, t\, \gamma} \in \mathbf{L}$
  - 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

$E \rightarrow T + E \mid T$

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

Entrada

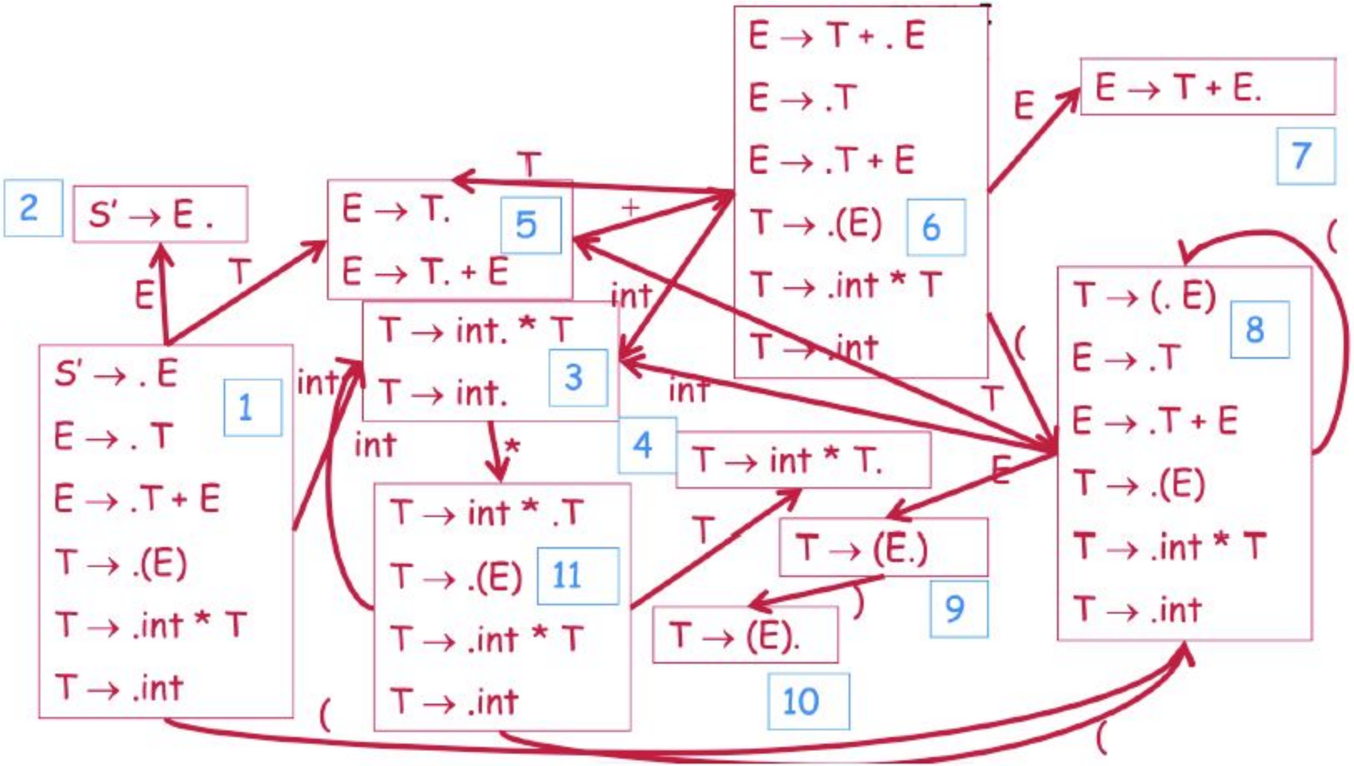
int * int

Símbolo de aceptación

$

Parsing SLR

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



Parsing SLR

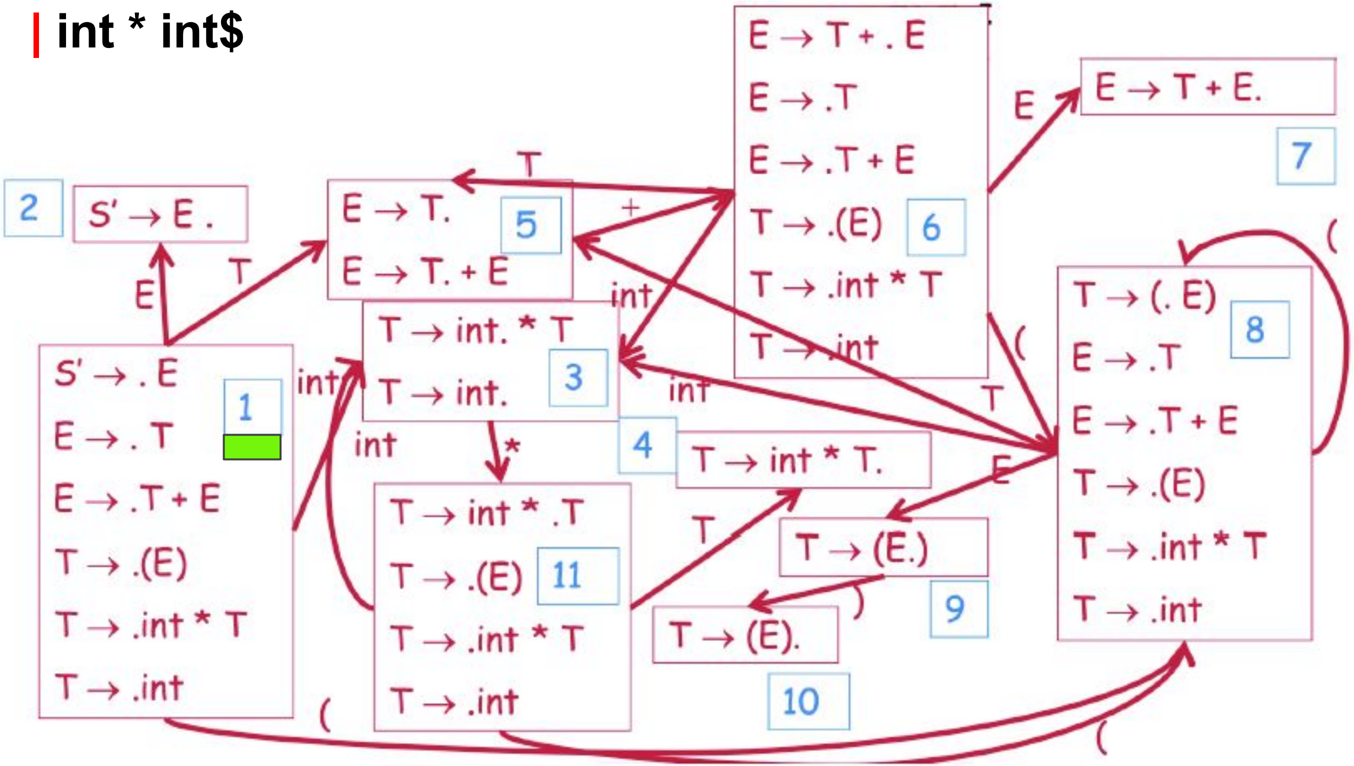
Follow(E) = { ‘)’, $ }
Follow(T) = { ‘+’, ‘)’, $ }

Entrada	Estado DFA	Acción
| int * int$	1	shift

Parsing SLR

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

int * int$



Parsing SLR

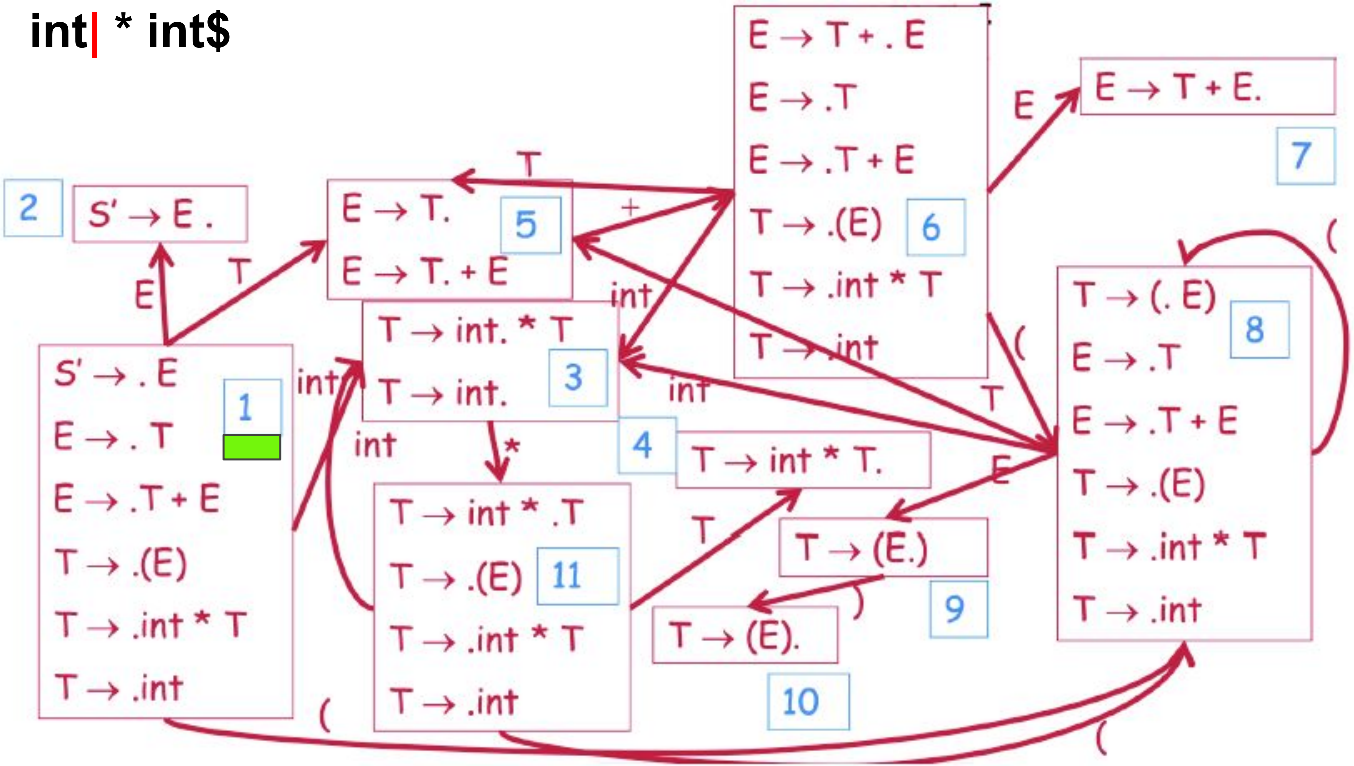
Follow(E) = { ‘)’, $ }
Follow(T) = { ‘+’, ‘)’, $ }

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

Parsing SLR

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

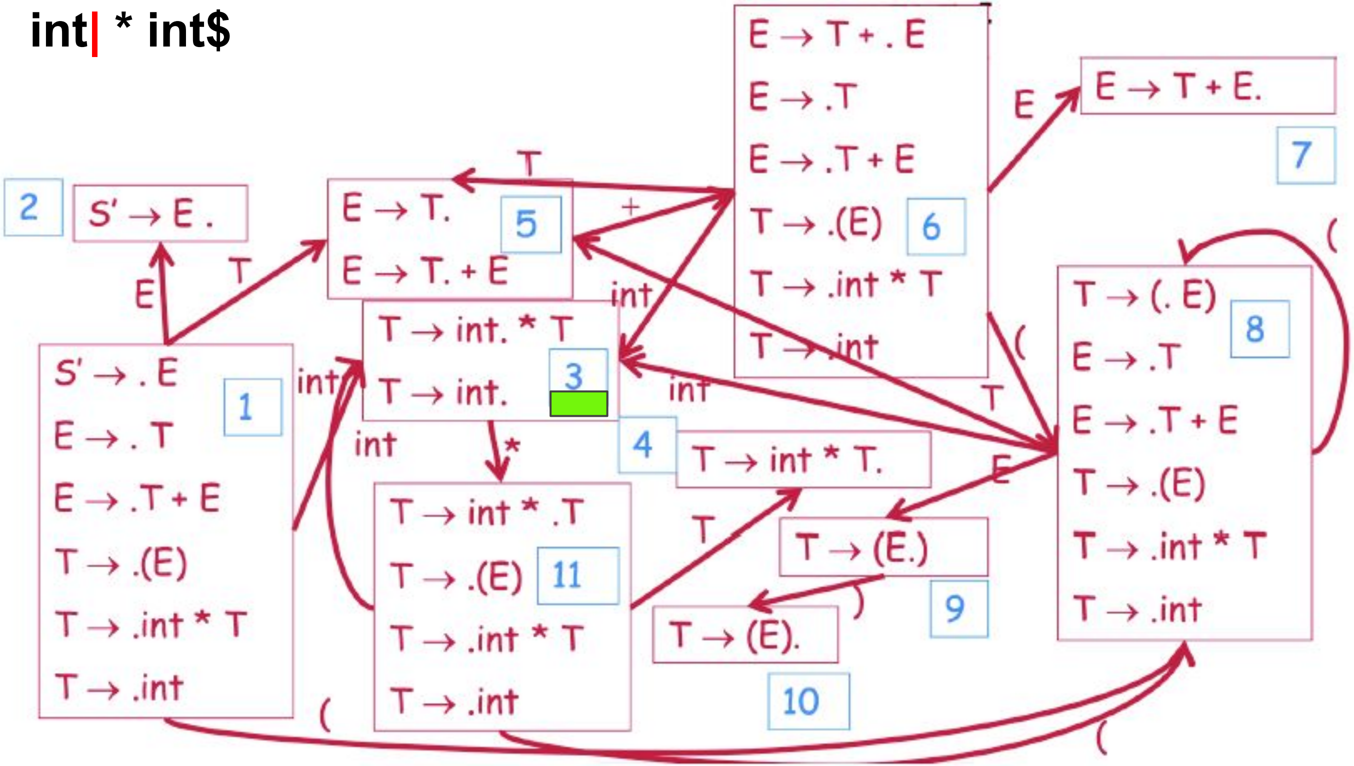
int| * int$



Parsing SLR

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

int| * int$



Parsing SLR

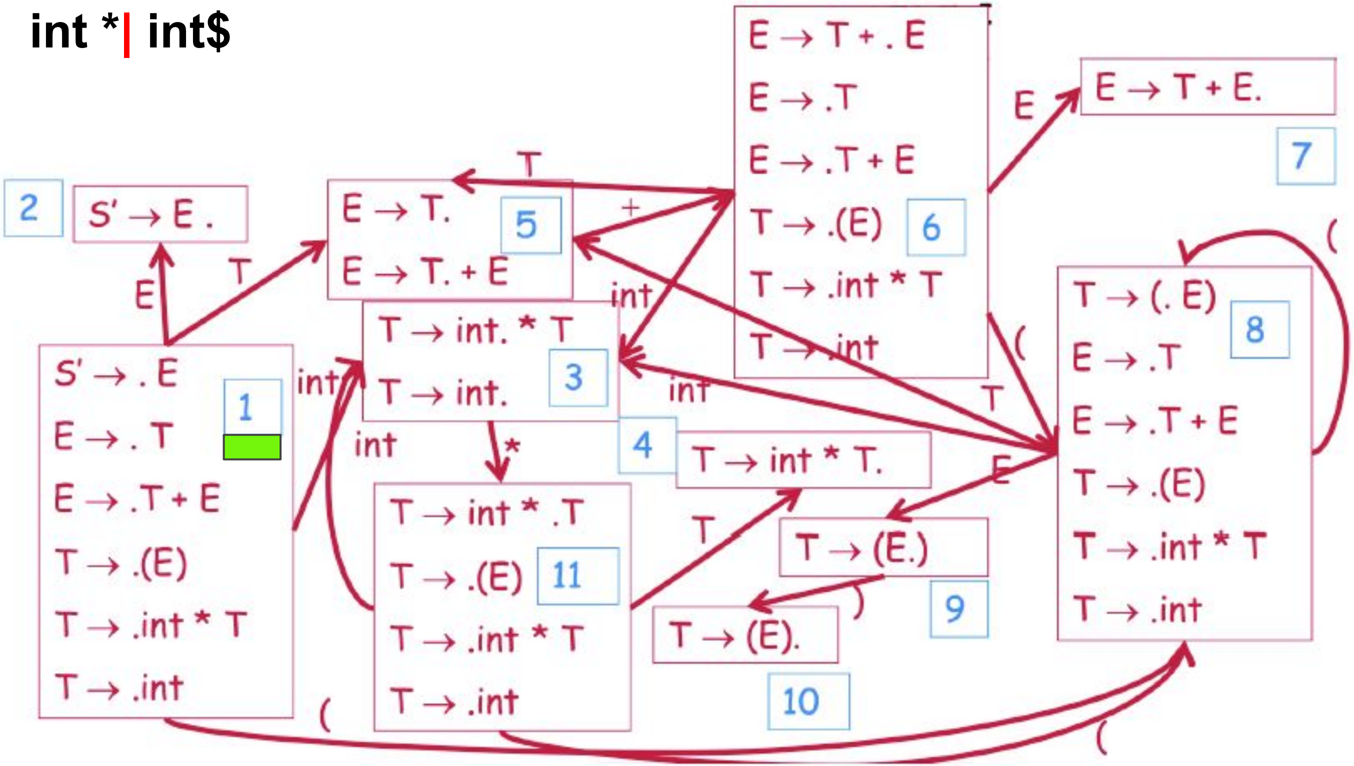
Follow(E) = { ‘)’, $ }
Follow(T) = { ‘+’, ‘)’, $ }

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

Parsing SLR

E → T + E | T
T → int * T | int | (E)

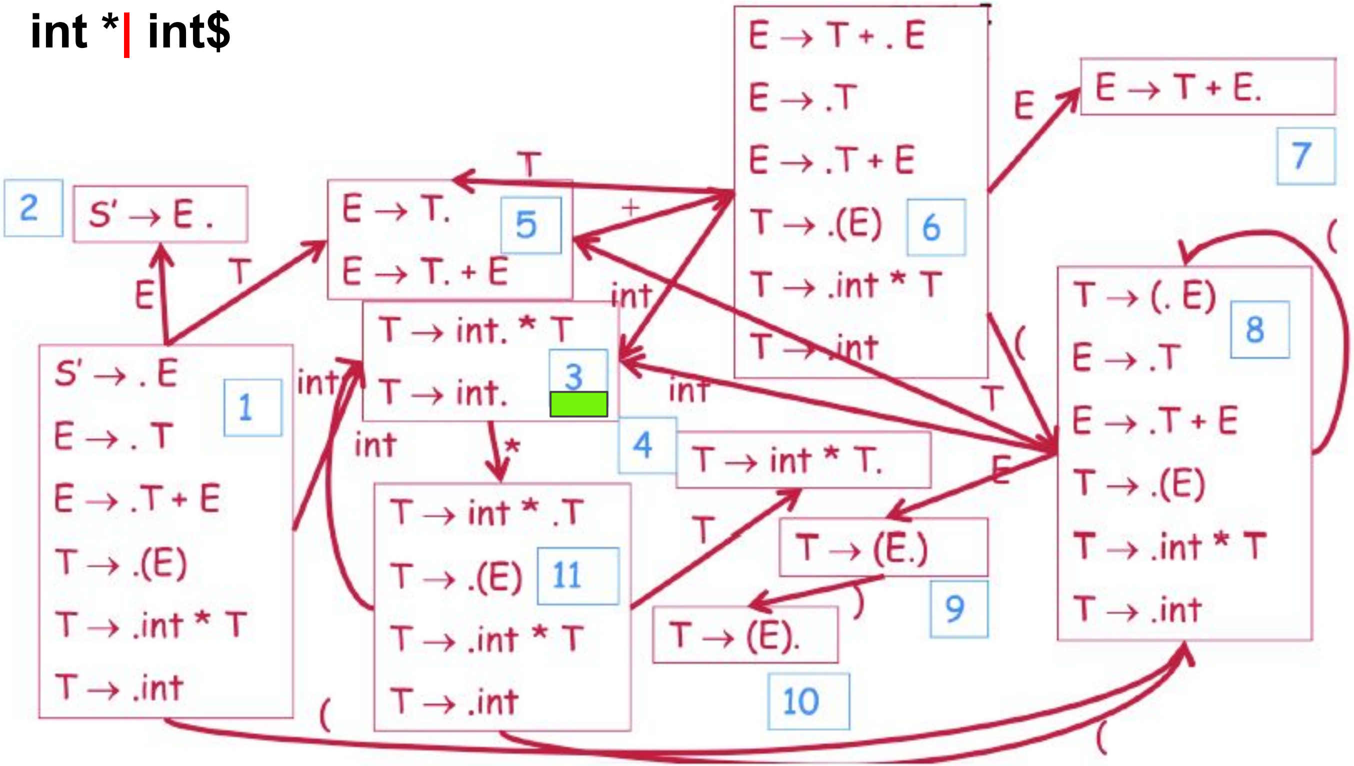
int *| int$



Parsing SLR

E → T + E | T
T → int * T | int | (E)

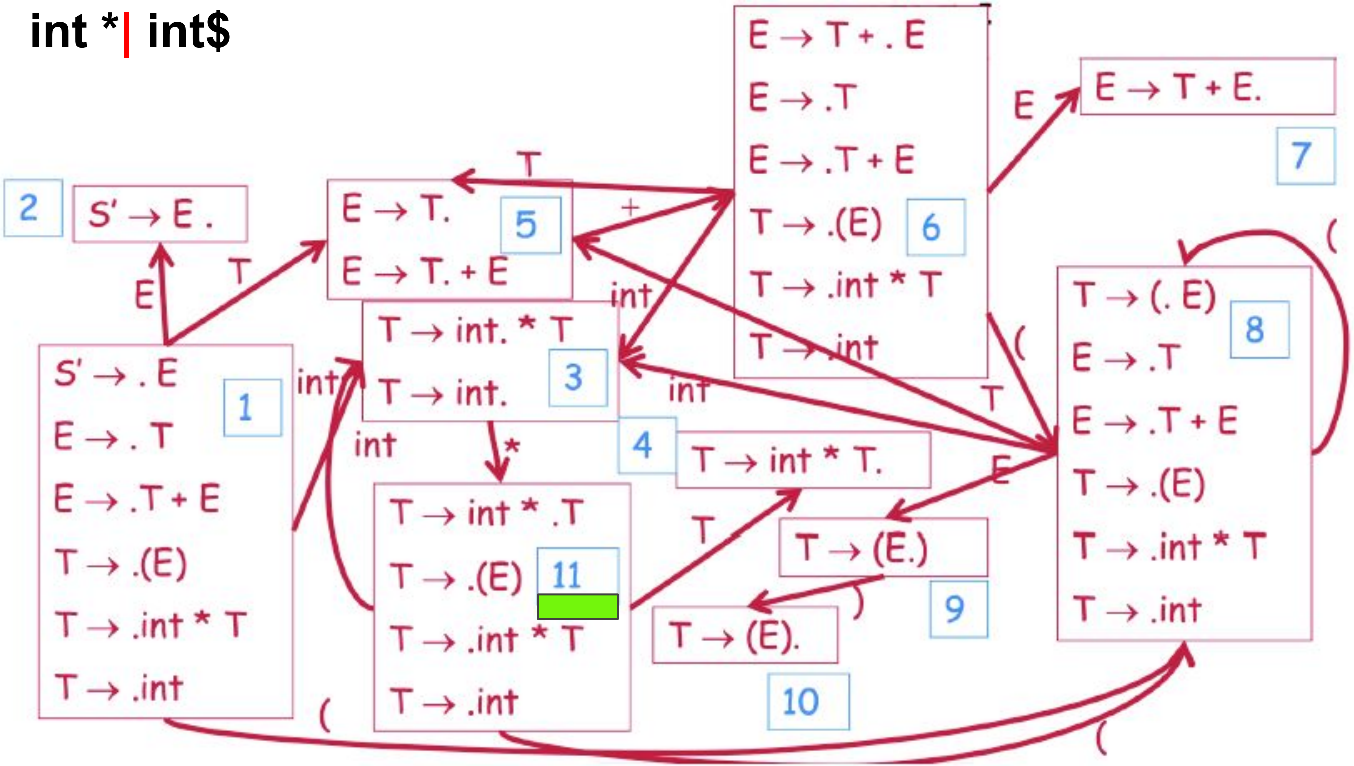
int *| int$



Parsing SLR

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

int *| int$



Parsing SLR

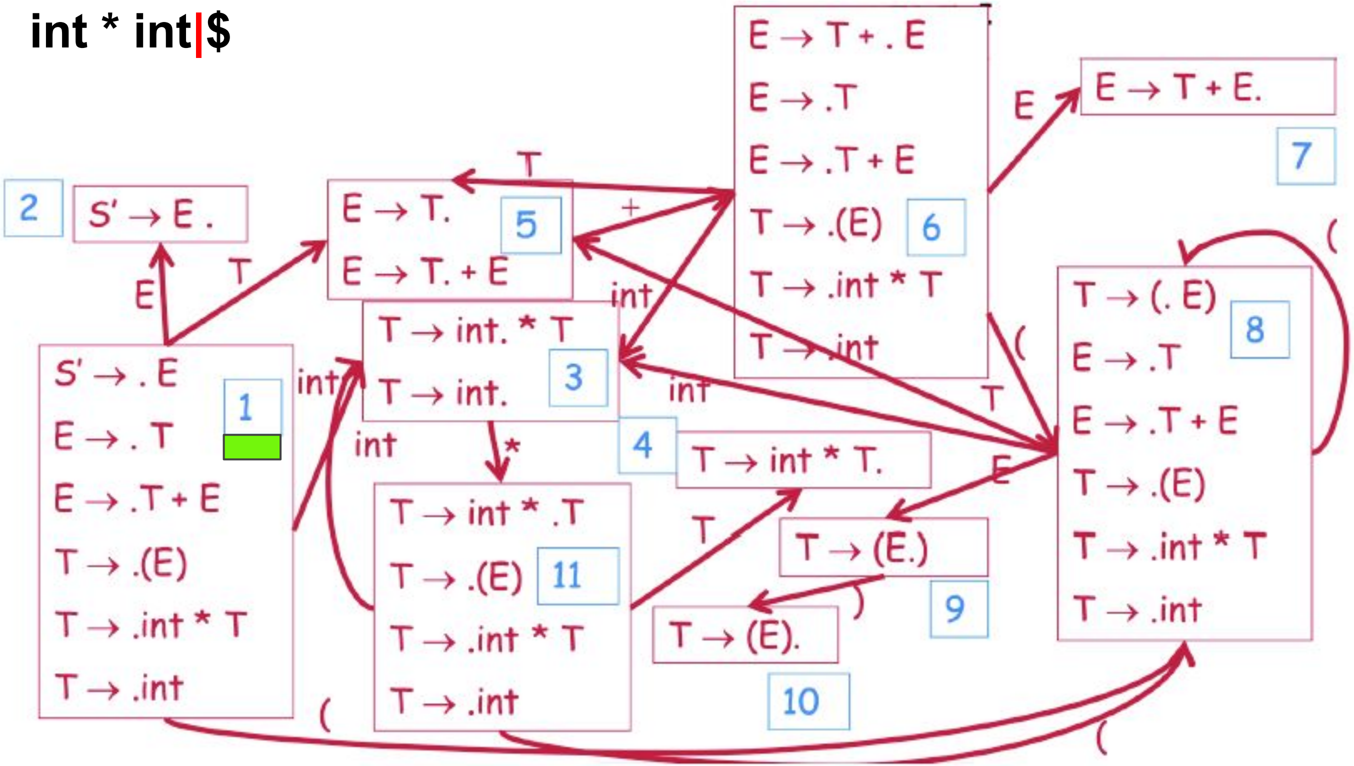
Follow(E) = { ‘)’, \$ }
Follow(T) = { ‘+’, ‘)’, \$ }

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

Parsing SLR

E → T + E | T
T → int * T | int | (E)

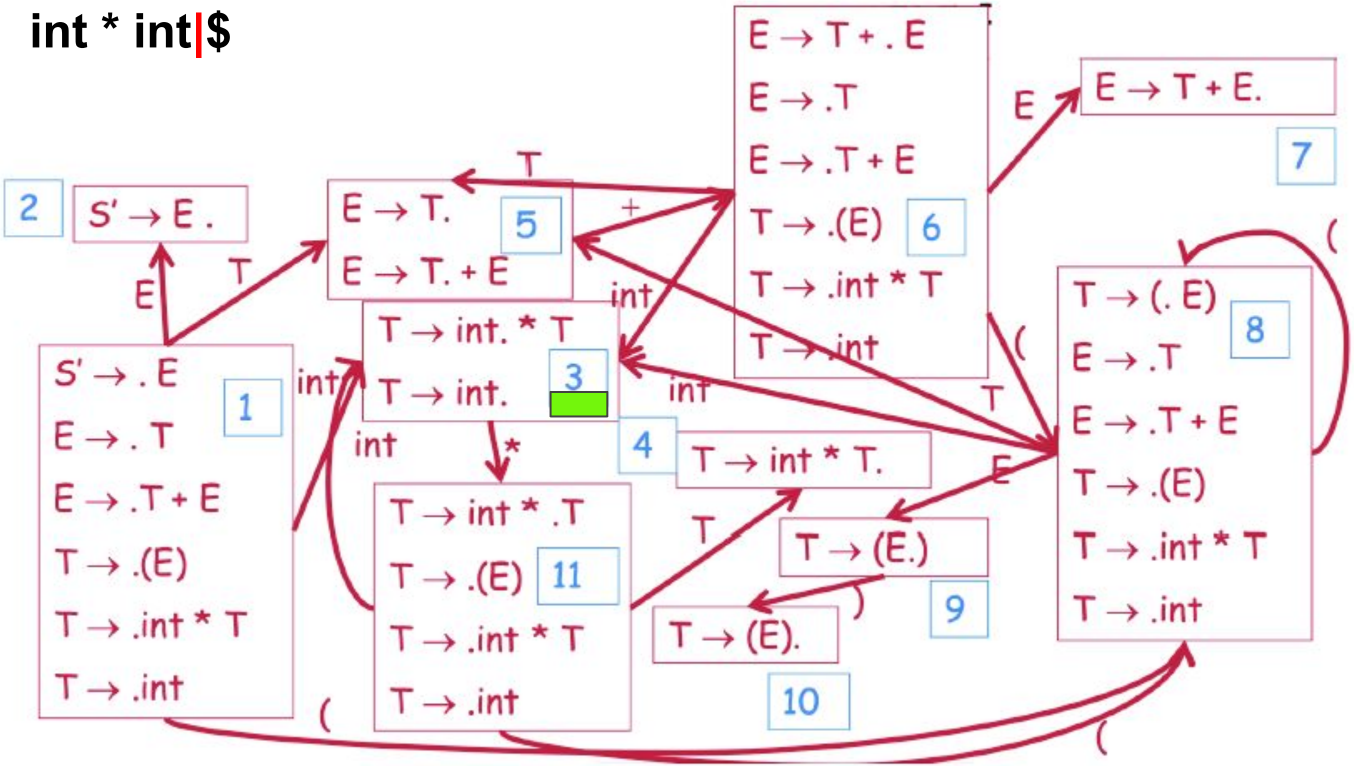
int * int|$



Parsing SLR

E → T + E | T
T → int * T | int | (E)

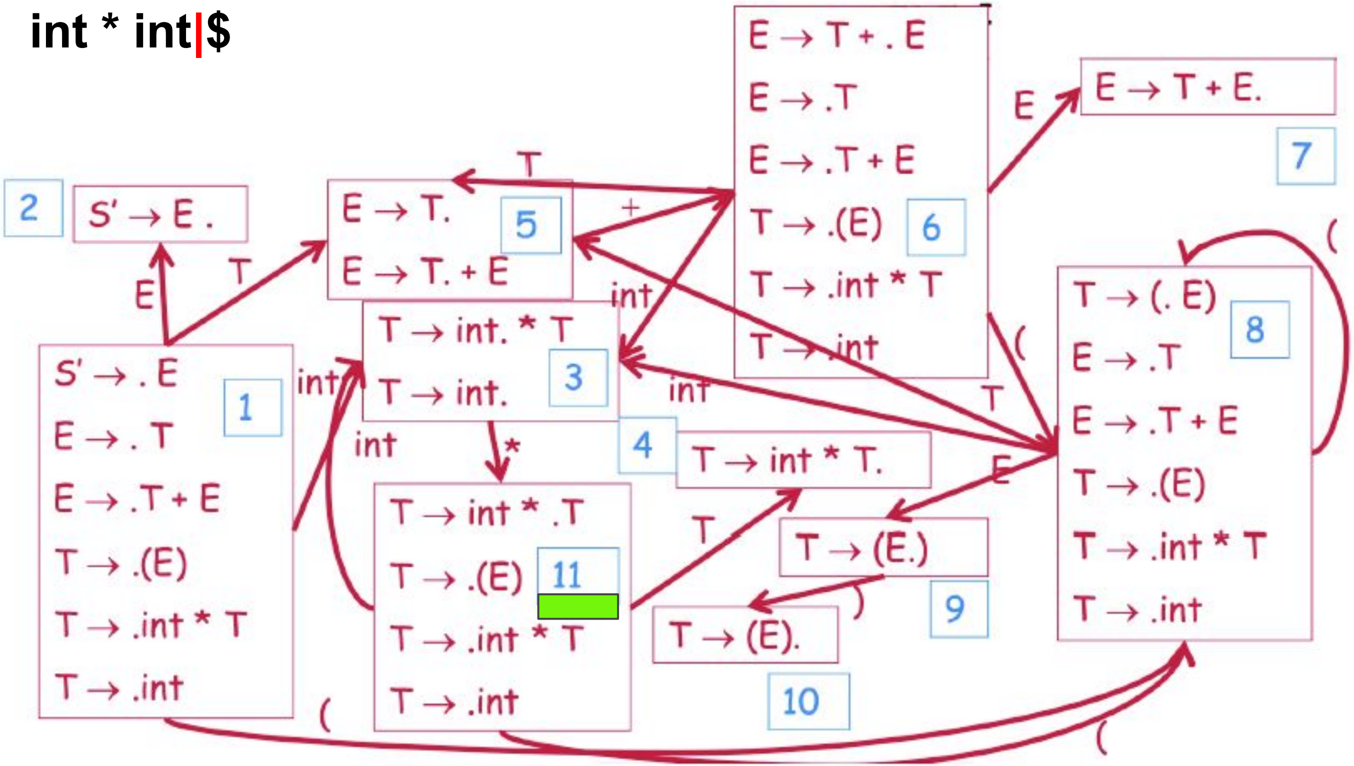
int * int|$



Parsing SLR

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

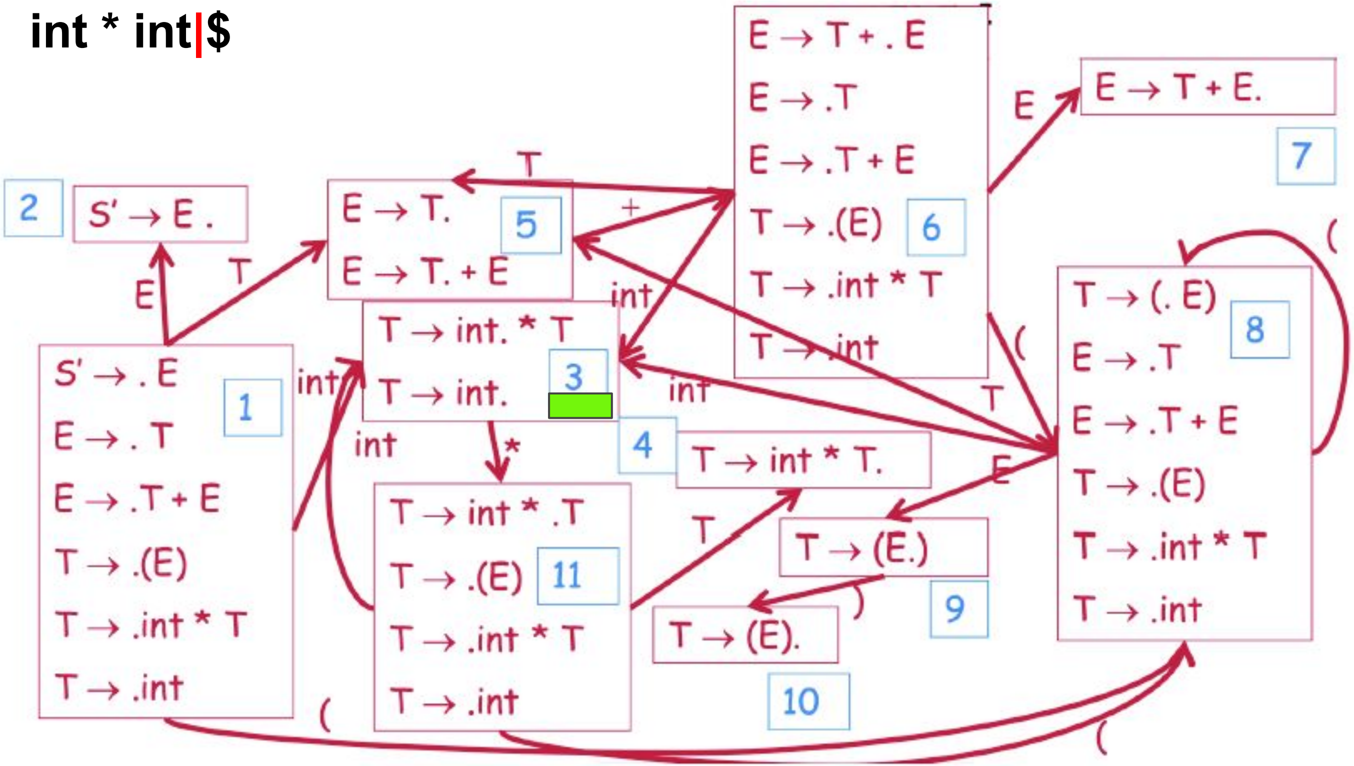
int * int|$



Parsing SLR

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

int * int|$



Parsing SLR

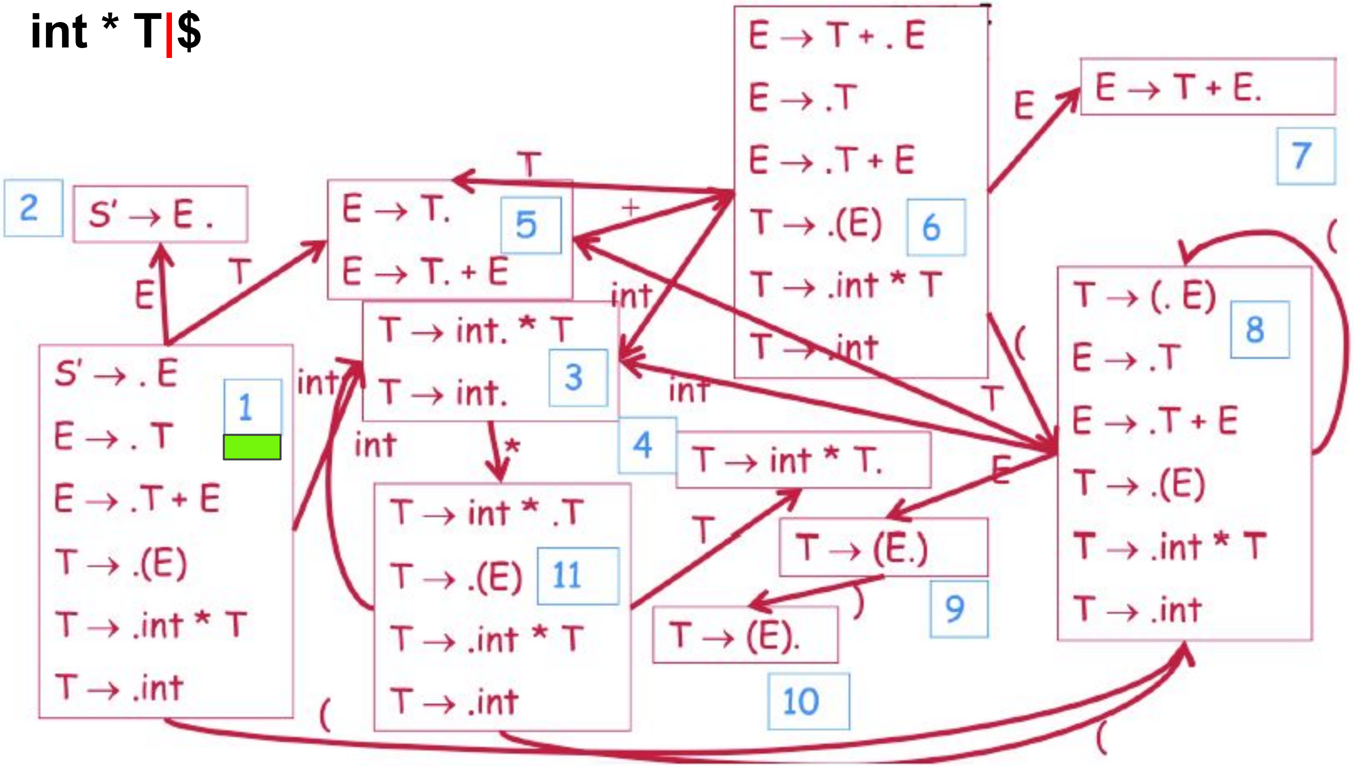
Follow(E) = { ')', $ }
Follow(T) = { '+', ')', $ }

Entrada	Estado DFA	Acción
| int * int$	1	shift
int |* int$	3 *** no es follow de T**	shift
int *| int$	11	shift
int * int|$	3 **$ ∈ Follow(T)**	reduce T→int
int * T|$	4 **$ ∈ Follow(T)**	reduce T→int*T

Parsing SLR

E → T + E | T
T → int * T | int | (E)

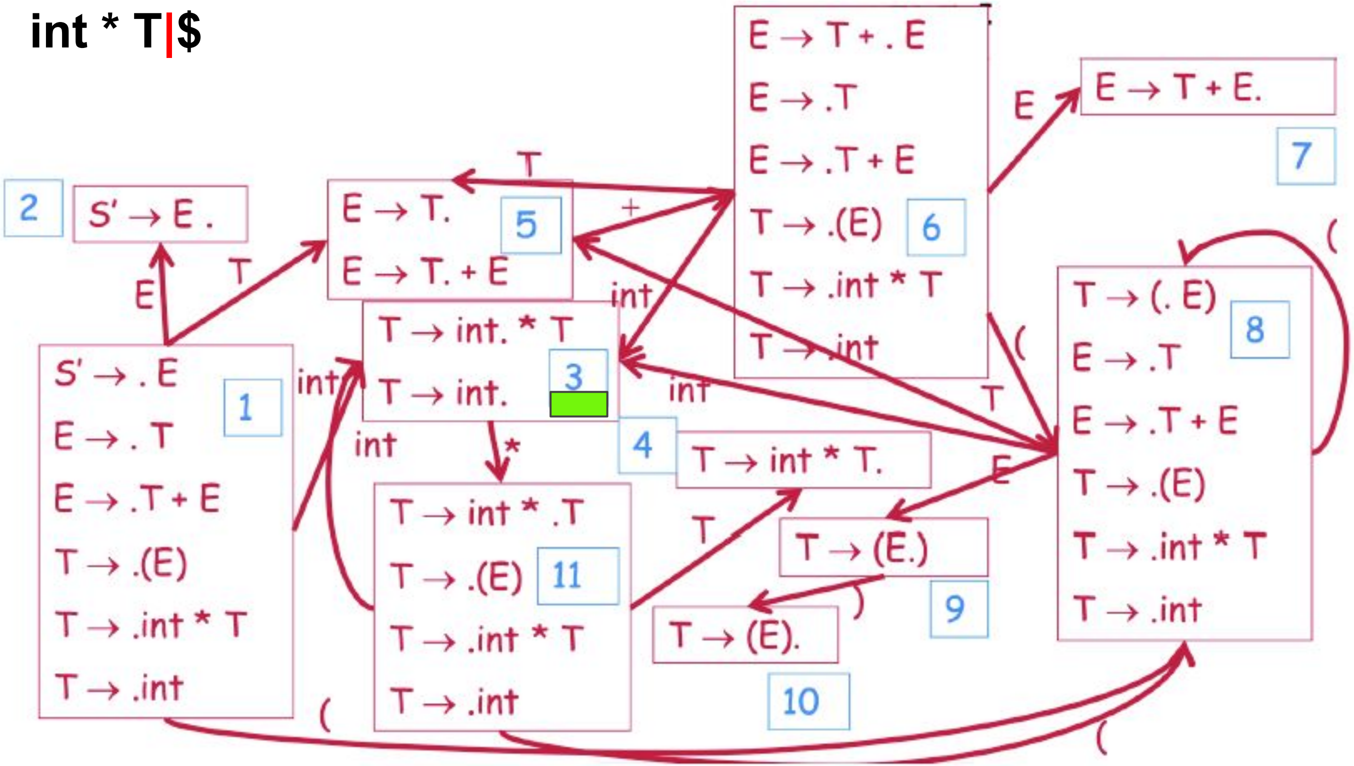
int * T|$



Parsing SLR

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

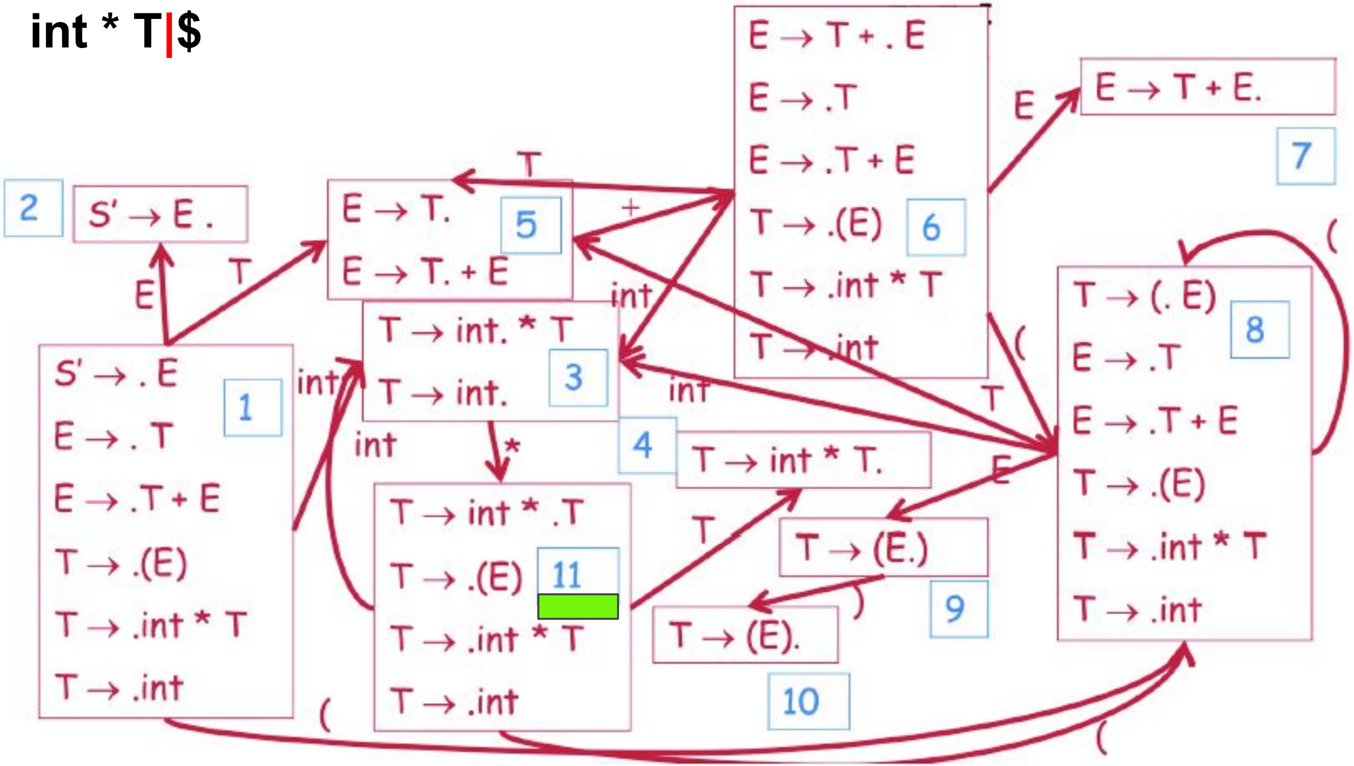
int * T|$



Parsing SLR

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

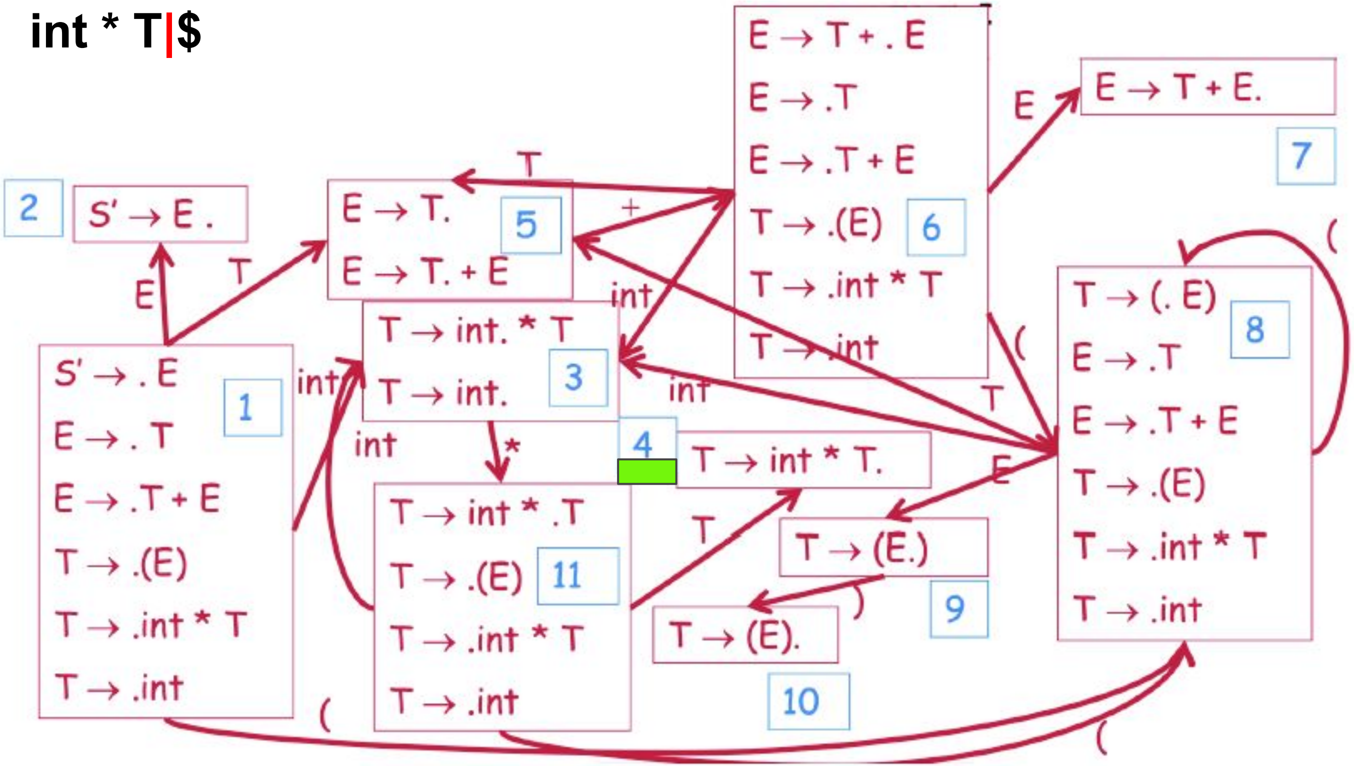
int * T|$



Parsing SLR

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

int * T|$



Parsing SLR

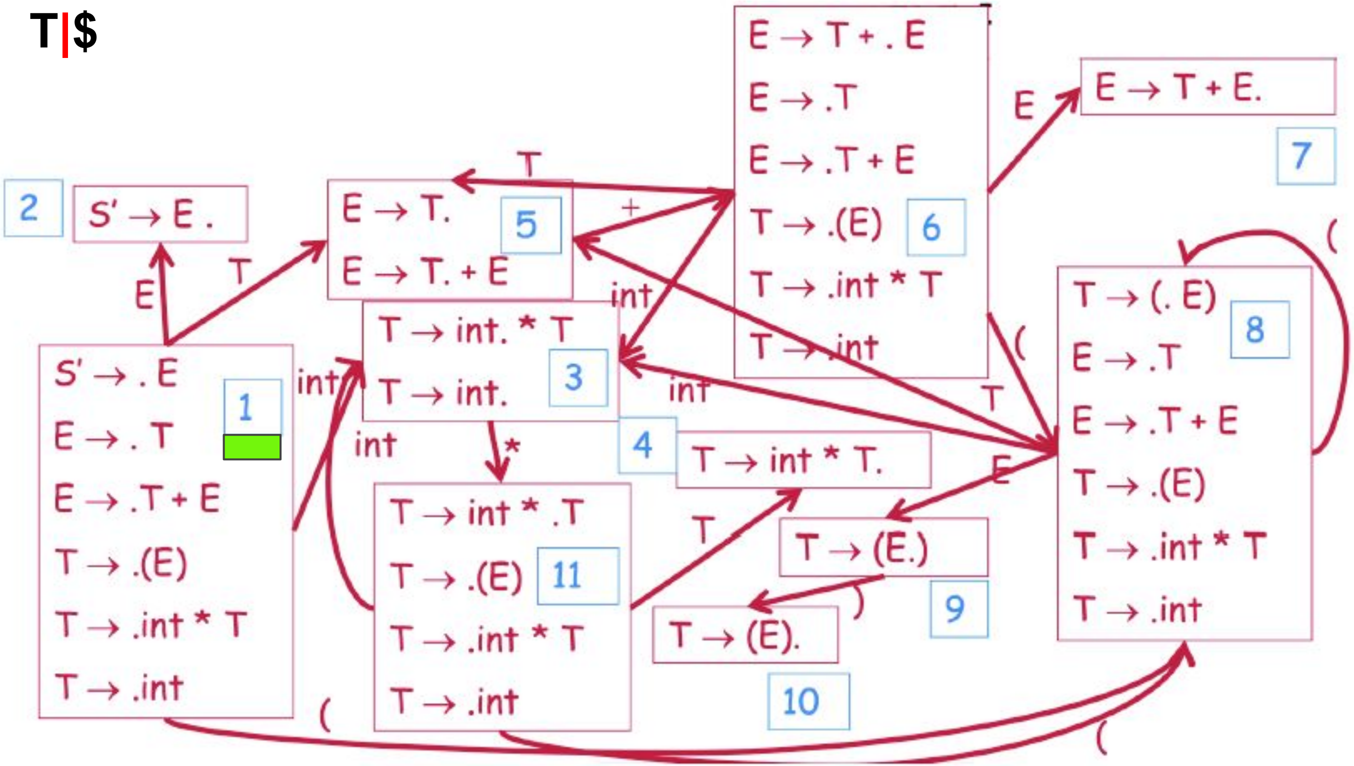
Follow(E) = { ‘)’, $ }
Follow(T) = { ‘+’, ‘)’, $ }

Entrada	Estado DFA	Acción
| int * int$	1	shift
int |* int$	3 *** no es follow de T**	shift
int *| int$	11	shift
int * int|$	3 **$ ∈ Follow(T)**	reduce T→int
int * T|$	4 **$ ∈ Follow(T)**	reduce T→int*T
T|$	5 **$ ∈ Follow(T)**	reduce E→T

Parsing SLR

E → T + E | T
T → int * T | int | (E)

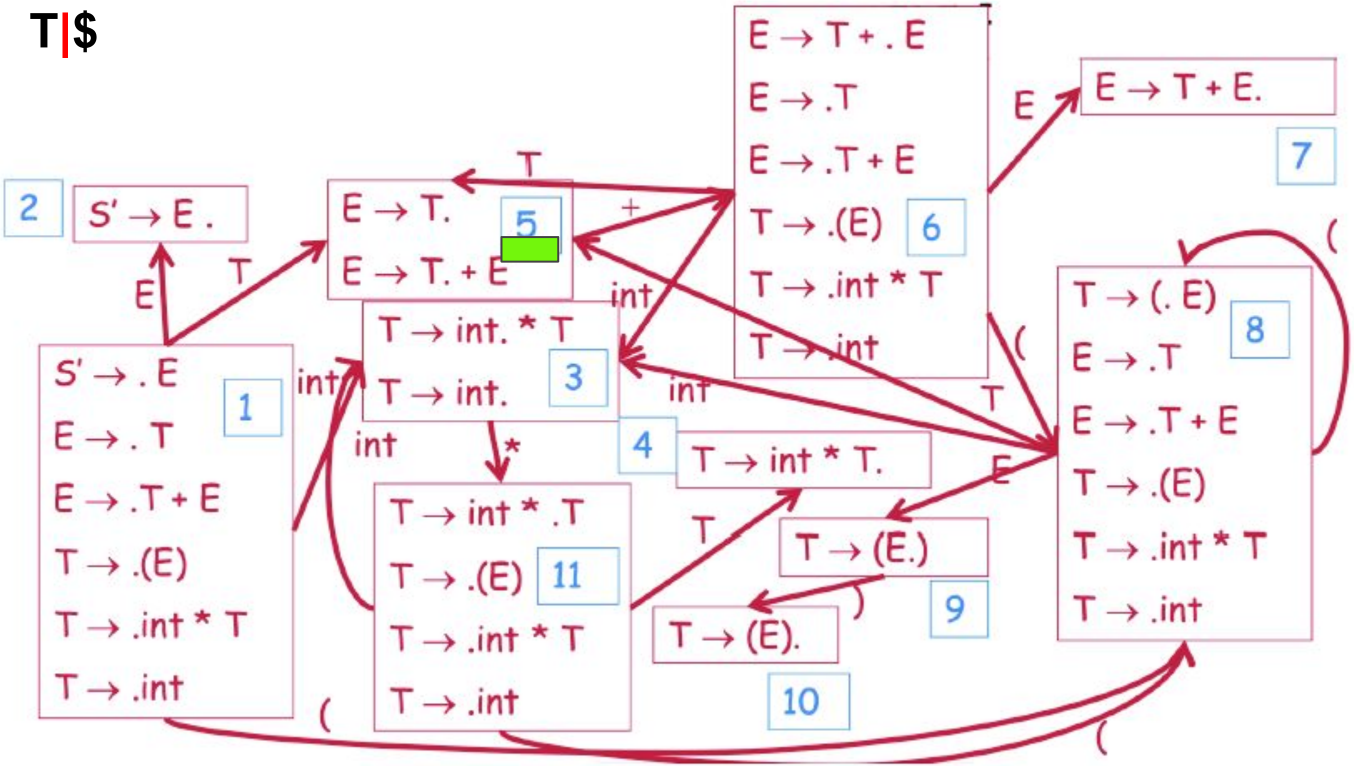
T|$



Parsing SLR

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

T|$



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
int * int|$	3 **$ ∈ Follow(T)**	reduce T→int
int * T|$	4 **$ ∈ Follow(T)**	reduce T→int*T
T|$	5 **$ ∈ Follow(T)**	reduce E→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
  ⟨ símbolo$_1$, estado$_1$ ⟩ . . . ⟨ símbolo$_n$, estado$_n$ ⟩,
  el ***estado$_n$*** es el estado final del DFA sobre **símbolo1...símbolo$_n$.**

- Detalle: La parte inferior del stack es ***⟨dummy, start⟩***, donde
  - ***dummy*** es un símbolo ficticio
  - ***start*** es el estado inicial del DFA.

Parsing SLR

Goto (DFA) Table

Definir **goto[i,A] = j** si **estado**$_i \rightarrow_A$ **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 **⟨a, x⟩** en la pila
  - **a** es la entrada actual
  - **x** es un estado del DFA
- Reduce **X** → **α**
  - 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 **goto[i,t] = k** entonces **action[i,t] = shift k**

– Si s_i tiene el item $X \rightarrow \alpha.$ and $t \in Follow(X)$ y $X \neq S'$ entonces

  action[i,t] = reduce $X \rightarrow \alpha$

– Si s_i tiene el item $S' \rightarrow S.$ then **action[i,$] = accept**

– De lo contrario, **action[i,t] = error**

Algoritmo SLR

```
Let input = w$ be initial input
Let j = 0
Let DFA state 1 be the one with item S' → .S
Let stack = ⟨ dummy, 1 ⟩  // ⟨ symbol, state ⟩
    repeat
        case action[top_state(stack), input[j]] of
            shift k:  push ⟨ input[j++], k ⟩
            reduce X → α:
                pop |α| 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
    - Después de ver **T→ int * T**, ***reduce*** si el **lookahead** es $.
  - Más preciso que simplemente usar conjuntos de follow.