Parsing (2)

15-411/15-611 Compiler Design

Seth Copen Goldstein

September, 29 2021

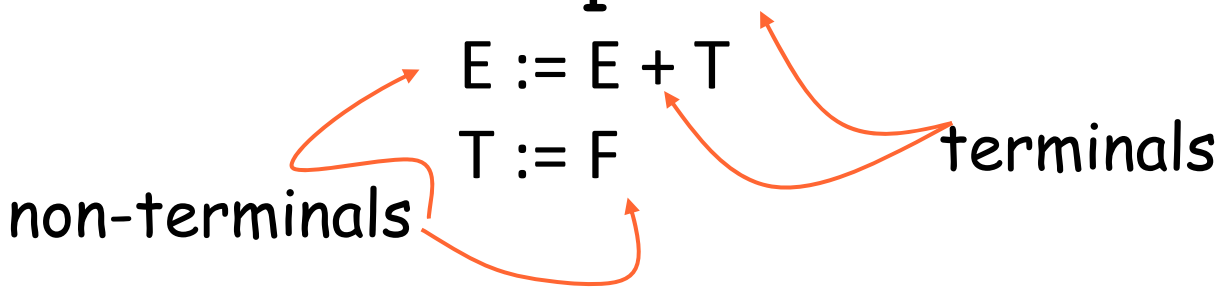
Today – Parsing Part 2

Parsing

- Top-down parsers
  - FIRST, FOLLOW, and NULLABLE
- Bottom-up parsers
  - handle pruning
  - parsing method
  - constructing state machine
  - LR0
  - SLR
  - LR(k) & LALR
  - Handling Ambiguity

Context-Free Grammar

- A context-free grammar, G, is described by:
  - Σ, a set of terminals …
  - A, a set of non-terminals.
  - S, S $\in$ A, the start symbol
  - P, set of productions …
    a production, p, has the form: : A$\rightarrow \alpha$
  - E.g.,: S := E
    S := **print** E
    E := E + T
    T := F



What makes a grammar CF?

- Only one NT on left-hand side $\rightarrow$ context-free
- What makes a grammar context-sensitive?
- $\alpha A \beta \rightarrow \alpha \gamma \beta$ where
  - α or β may be empty,
  - but γ is not-empty
- Are context-sensitive grammars useful for compiler writers?

Simple Grammar of Expressions

S := Exp

Exp := Exp + Exp

Exp := Exp - Exp

Exp := Exp * Exp

Exp := Exp / Exp

Exp := **`id`**

Exp := **`int`**

Describes a language of expressions. e.g.: 2+3*x

Leftmost Derivations

- Leftmost derivation: leftmost NT always chosen

```
1  S   := Exp
2  Exp:= Exp + Exp
3  Exp:= Exp - Exp
4  Exp:= Exp * Exp
5  Exp:= Exp / Exp
6  Exp:= id
7  Exp:= int
```

S

by 1 $\Rightarrow$ Exp

by 4 $\Rightarrow$ Exp * Exp

by 2 $\Rightarrow$ Exp + Exp * Exp

by 7 $\Rightarrow$ $\mathtt{int}_2$ + Exp * Exp

by 7 $\Rightarrow$ $\mathtt{int}_2$ + $\mathtt{int}_3$ * Exp

by 6 $\Rightarrow$ $\mathtt{int}_2$ + $\mathtt{int}_3$ * $\mathtt{id}_x$

Rightmost Derivations

- Rightmost derivation: rightmost NT always chosen

```
1  S   := Exp
2  Exp:= Exp + Exp
3  Exp:= Exp - Exp
4  Exp:= Exp * Exp
5  Exp:= Exp / Exp
6  Exp:= id
7  Exp:= int
```

S

by 1 $\Rightarrow$ Exp

by 4 $\Rightarrow$ Exp * Exp

by 6 $\Rightarrow$ Exp * $\mathtt{id}_x$

by 2 $\Rightarrow$ Exp + Exp * $\mathtt{id}_x$

by 7 $\Rightarrow$ Exp + $\mathtt{int}_3$ * $\mathtt{id}_x$

by 7 $\Rightarrow$ $\mathtt{int}_2$ + $\mathtt{int}_3$ * $\mathtt{id}_x$

Parse Trees

- symbols in rhs are children of NT being rewritten

S

by 1 ⇒ Exp

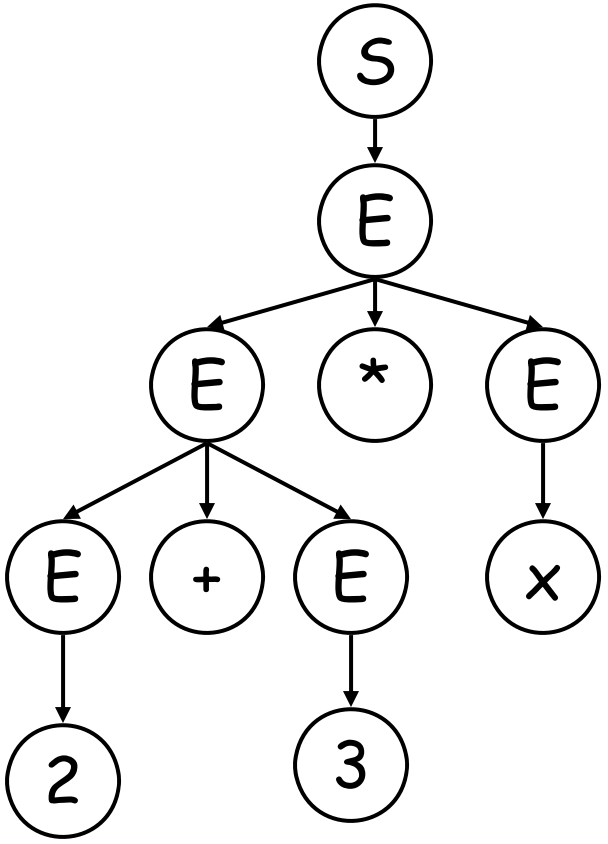
by 4 ⇒ Exp * Exp

by 2 ⇒ Exp + Exp * Exp

by 7 ⇒ $\texttt{int}_2$ + Exp * Exp

by 7 ⇒ $\texttt{int}_2$ + $\texttt{int}_3$ * Exp

by 6 ⇒ $\texttt{int}_2$ + $\texttt{int}_3$ * $\texttt{id}_x$



Converting Expression Grammar

- Adding precedence with more non-terminals
- One for each level of precedence:
  - (+, -) exp
  - (*, /) term
  - (**`id`**, **`int`**) factor
  - Make sure parse derives sentences that respect the precedence
  - Make sure that extra levels of precedence can be bypassed, i.e., “x” is still legal

A Better Exp Grammar

1 S := Exp
2 Exp := Exp + Term
3 Exp := Exp - Term
4 Exp := Term
5 Term := Term * Factor
6 Term := Term / Factor
7 Term := Factor
8 Factor := **id**
9 Factor := **int**

S
by 1 ⇒ Exp
by 2 ⇒ Exp + Term
by 4 ⇒ Term + Term
by 7 ⇒ Factor + Term
by 9 ⇒ $\mathbf{int}_2$ + Term
by 5 ⇒ $\mathbf{int}_2$ + Term * Factor
by 7 ⇒ $\mathbf{int}_2$ + Factor * Factor
by 9 ⇒ $\mathbf{int}_2$ + $\mathbf{int}_3$ * Factor
by 8 ⇒ $\mathbf{int}_2$ + $\mathbf{int}_3$ * $\mathbf{id}_x$

What is the parse tree?

Parsing a CFG

- Top-Down
  - start at root of parse-tree
  - pick a production and expand to match input
  - may require backtracking
  - if no backtracking required, predictive
- Bottom-up
  - start at leaves of tree
  - recognize valid prefixes of productions
  - consume input and change state to match
  - use stack to track state

Top-down Parsers

- Starts at root of parse tree and recursively expands children that match the input
- In general case, may require backtracking
- Such a parser uses recursive descent.
- When a grammar does not require backtracking a predictive parser can be built.

Top-Down parsing

- Start with root of tree, i.e., S
- Repeat until entire input matched:
  - pick a non-terminal, A, and pick a production $A \rightarrow \gamma$ that can match input, and expand tree
  - if no such rule applies, backtrack
- Key is obviously selecting the right production

Top-down for Exp Grammar

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := **id**
9	F := **int**

	S	$\mathbf{int}_2$ - $\mathbf{int}_3$ * $\mathbf{id}_x$
by 1 ⇒	E	$\mathbf{int}_2$ - $\mathbf{int}_3$ * $\mathbf{id}_x$

Top-down for Exp Grammar

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := id
9	F := int

S	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 1 ⇒ E	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 2 ⇒ E + T	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 4 ⇒ T + T	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 7 ⇒ F + T	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 9 ⇒ $\texttt{int}_2$ + T	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$

Must backtrack here!

Top-down for Exp Grammar

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := id
9	F := int

S	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 1 ⇒ E	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 2 ⇒ E + T	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 4 ⇒ T + T	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 7 ⇒ F + T	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 9 ⇒ $\texttt{int}_2$ + T	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 3 ⇒ E - T	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 4 ⇒ T - T	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 7 ⇒ F - T	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 9 ⇒ $\texttt{int}_2$ - T	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 5 ⇒ $\texttt{int}_2$ - T * F	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$

Top-down for Exp Grammar

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := **id**
9	F := **int**

	S	**int**$_2$ - **int**$_3$ * **id**$_x$
by 1 ⇒	E	**int**$_2$ - **int**$_3$ * **id**$_x$
by 2 ⇒	E + T	**int**$_2$ - **int**$_3$ * **id**$_x$
by 4 ⇒	T + T	**int**$_2$ - **int**$_3$ * **id**$_x$
by 7 ⇒	F + T	**int**$_2$ - **int**$_3$ * **id**$_x$
by 9 ⇒	**int**$_2$ + T	**int**$_2$ - **int**$_3$ * **id**$_x$
by 3 ⇒	E - T	**int**$_2$ - **int**$_3$ * **id**$_x$
by 4 ⇒	T - T	**int**$_2$ - **int**$_3$ * **id**$_x$
by 7 ⇒	F - T	**int**$_2$ - **int**$_3$ * **id**$_x$
by 9 ⇒	**int**$_2$ - T	**int**$_2$ - **int**$_3$ * **id**$_x$
		int$_2$ - **int**$_3$ * **id**$_x$

What kind of derivation is this parsing?

Top-down for Exp Grammar

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := **id**
9	F := **int**

	S	**int**$_2$ - **int**$_3$ * **id**$_x$
by 1 ⇒	E	**int**$_2$ - **int**$_3$ * **id**$_x$
by 2 ⇒	E + T	**int**$_2$ - **int**$_3$ * **id**$_x$
by 2 ⇒	E + E + T	**int**$_2$ - **int**$_3$ * **id**$_x$
by 2 ⇒	E + E + E + T	**int**$_2$ - **int**$_3$ * **id**$_x$

Will not terminate! Why?

grammar is left-recursive

What should we do about it?

Eliminate left-recursion

Eliminating Left-Recursion

- Given 2 productions:

  $A := A\ \alpha \mid \beta$

  Where neither α nor β start with A

  (e.g., For example, E := E + T | T)

  α β

- Make it right-recursive:

  $A := \beta\ R$

  $R := \alpha\ R$

  $\mid$

  R is right recursive

- Extends to general case.

Rewriting Exp Grammar

1	S	:= E
2	E	:= E + T
3	E	:= E - T
4	E	:= T
5	T	:= T * F
6	T	:= T / F
7	T	:= F
8	F	:= **id**
9	F	:= **int**

1	S	:= E
2′	E′	:= + T E′
3′	E′	:= - T E′
4′	E′	:=
5′	T′	:= * F T′
6′	T′	:= / F T′
7′	T′	:=
8	F	:= **id**
9	F	:= **int**

Is this legible?

2	E	:= T E′
5	T	:= F T′

Try again

1	S := E
2	E := T E′
2′	E′ := + T E′
3′	E′ := - T E′
4′	E′ :=
5	T := F T′
5′	T′ := * F T′
6′	T′ := / F T′
7′	T′ :=
8	F := id
9	F := int

S	●int$_2$ - int$_3$ * id$_x$
by 1 ⇒ E	●int$_2$ - int$_3$ * id$_x$
by 2 ⇒ T E′	●int$_2$ - int$_3$ * id$_x$
by 5 ⇒ F T′ E′	●int$_2$ - int$_3$ * id$_x$
by 9 ⇒ 2 T′ E′	int$_2$ ●- int$_3$ * id$_x$
by 7′ ⇒ 2 E′	int$_2$ ●- int$_3$ * id$_x$
by 3′ ⇒ 2 - T E′	int$_2$ - ●int$_3$ * id$_x$
by 5 ⇒ 2 - F T′ E′	int$_2$ - ●int$_3$ * id$_x$
by 9 ⇒ 2 - 3 T′ E′	int$_2$ - int$_3$ ●* id$_x$
by 5′ ⇒ 2 - 3 * F T′ E′	int$_2$ - int$_3$ * ●id$_x$
	int$_3$ * id$_x$●
	int$_3$ * id$_x$●
	int$_3$ * id$_x$●

Unlike previous time we tried this, it appears that only one production applies at a time. I.e., no backtracking needed. Why?

Lookahead

- How to pick right production?
- Lookahead in input stream for guidance
- General case: arbitrary lookahead required
- Luckily, many context-free grammars can be parsed with limited lookahead
- If we have $A \rightarrow \alpha \mid \beta$, then we want to correctly choose either $A \rightarrow \alpha$ or $A \rightarrow \beta$
- define FIRST(α) as the set of tokens that can be first symbol of α, i.e.,

  $a \in$ FIRST(α) iff $\alpha \rightarrow^* a\gamma$ for some γ

Lookahead

- How to pick right production?
- If we have $A \rightarrow \alpha \mid \beta$, then we want to correctly choose either $A \rightarrow \alpha$ or $A \rightarrow \beta$
- define FIRST(α) as the set of tokens that can be first symbol of α, i.e.,

  $a \in$ FIRST(α) iff $\alpha \rightarrow^* a\gamma$ for some γ
- If $A \rightarrow \alpha \mid \beta$ we want:

  FIRST(α) $\cap$ FIRST(β) = $\varnothing$
- If that is always true, we can build a predictive parser.

Computing FIRST(α)

- Given X := A B C, FIRST(X) = FIRST(A B C)
- Can we ignore B or C?
- Consider:

```
A := a
   |
B := b
   | A
C := c
```

Computing FIRST(α)

- Given X := A B C, FIRST(X) = FIRST(A B C)
- Can we ignore B or C?
- Consider:

```
A := a
   |
B := b
   | A
C := c
```

- FIRST(X) must also include FIRST(C)
- IOW:
  - Must keep track of NTs that are nullable
  - For nullable NTs, determine FOLLOWS(NT)

nullable(A)

- nullable(A) is
  - true if A can derive the empty string
  - false otherwise
- For example:

```
B := X Y b
X := x
   |  Y Y
Y :=
```

In this case, nullable(X) = nullable(Y) = true

nullable(B) = false

FOLLOW(A)

- FOLLOW(A) is the set of terminals that can immediately follow A in a sentential form.
- I.e.,
  a $\in$ FOLLOW(A) iff S $\Rightarrow^*$ αAaβ for some α and β

Building a Predictive Parser

- We want to know for each non-terminal which production to choose based on the next input character.
- Build a table with rows labeled by non-terminals, A, and columns labeled by terminals, a. We will put the production, A := α , in (A, a) iff
  - FIRST(α) contains a or
  - nullable(α) and FOLLOW(A) contains a

The table for the robot

S := B S F

|

B := b

F := f

	FIRST	FOLLOW	nullable
S	b	$	yes
B	b	b,f	no
F	f	f,$	no

	b	f	$
S			
B			
F			

The table for the robot

S := B S F

|

B := b

F

	FIRST	FOLLOW	nullable
S	b	$	yes
B	b	b,f	no
F	f	f,$	no

FIRST(BSF) = b

nullable(ε)=true and FOLLOW(S) = $

	b	f	$
S	S:=BSF		S:=
B	B:=b		
F		F:=f	

Table f

```
1    S := E
2    E := T E'
2'   E' := + T E'
3'   E' := - T E'
4'   E' :=
5    T := F T'
5'   T' := * F T'
6'   T' := / F T'
7'   T' :=
8    F := id
9    F := int
```

	FIRST	FOLLOW	nullable
S	id, int	$	
E	id, int	$	
E'	+, -	$	yes
T	id, int	+,-,$	
T'	/, *	+,-,$	yes
F	id, int	/, *,$	

	+	-	*	/	id	int	$
S							
E							
E'							
T							
T'							
F							

Table f

1	S := E
2	E := T E'
2'	E' := + T E'
3'	E' := - T E'
4'	E' :=
5	T := F T'
5'	T' := * F T'
6'	T' := / F T'
7'	T' :=
8	F := id
9	F := int

	FIRST	FOLLOW	nullable
S	id, int	$	
E	id, int	$	
E'	+, -	$	yes
T	id, int	+,-,$	
T'	/, *	+,-,$	yes
F	id, int	/, *,$	

	+	-	*	/	id	int	$
S					:=E	:=E	
E					:=TE'	:=TE'	
E'	:=+TE'	:=-TE'					:=
T					:=FT'	:=FT'	
T'	:=	:=	:=*FT'	:=/FT'			:=
F					:=id	:=int	

Using the Table

- Each row in the table becomes a function
- For each input token with an entry: Create a series of invocations that implement the production, where
  - a non-terminal is eaten
  - a terminal becomes a recursive call
- For the blank cells implement errors

Example function

	+	-	*	/	id	int	$
S					:=E	:=E	
E					:=TE'	:=TE'	
E'	:=+TE'	:=-TE'			:=TE'	:=TE'	:=
T							
T'	:=	:=	:=*FT				
F					:=id	:=int	

How to handle errors?

```
Eprime() {
   switch (token) {
   case PLUS:     eat(PLUS); T(); Eprime(); break;
   case MINUS:    eat(MINUS); T(); Eprime(); break;
   case ID:       T(); Eprime();
   case INT:      T(); Eprime();
   default:       error();
  }
```

Left-Factoring

- Predictive parsers need to make a choice based on the next terminal.
- Consider:

```
S := if E then S else S
   | if E then S
```

- When looking at **`if`**, can't decide
- so left-factor the grammar

```
S := if E then S X
X := else S
   |
```

Top-Down Parsing

- Can be constructed by hand
- LL(k) grammars can be parsed
  - Left-to-right
  - Leftmost-derivation
  - with k symbols lookahead
- Often requires
  - left-factoring
  - Elimination of left-recursion

Bottom-up parsers

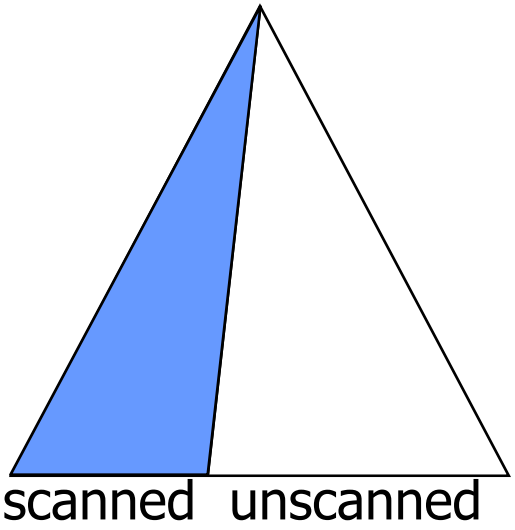
- What is the inherent restriction of top-down parsing, e.g., with LL(k) grammars?

Bottom-up parsers

- What is the inherent restriction of top-down parsing, e.g., with LL(k) grammars?
- Bottom-up parsers use the entire right-hand side of the production
- LR(k):
  - Left-to-right parse,
  - Rightmost derivation (in reverse),
  - k look ahead tokens

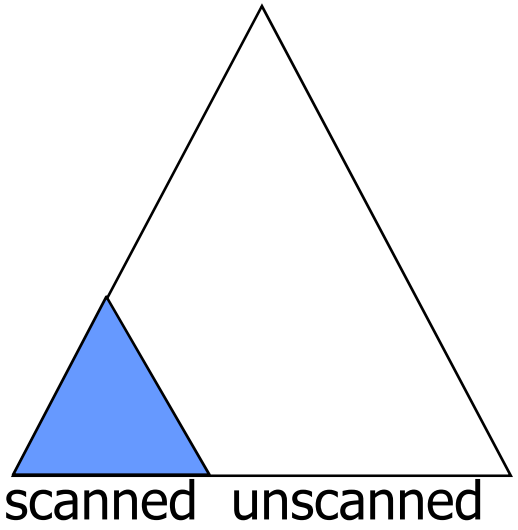
Top-down vs. Bottom-up

LL(k), recursive descent



Top-down

LR(k), shift-reduce



Bottom-up

Example - Top-down

```
S := X
X := X a
   | b
```

Is this grammar LL(k)?

How can we make it LL(k)?

```
S := X
X := b R
R := a R
   |
```

What about a bottom up parse?

Example - Bottom-up

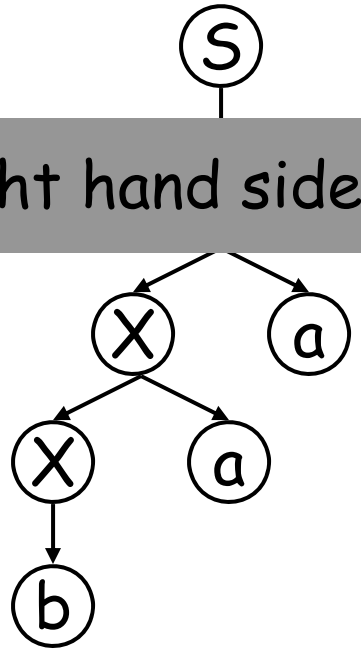
S := X
X := X a
| b

right-most derivation:

LR parser gets to look at an entire right hand side.

Left-to-Right, Rightmost in reverse

baa
Xaa
Xa
X
S



A Rightmost Derivation

1 S := Exp
2 Exp := Exp + Term
3 Exp := Exp - Term
4 Exp := Term
5 Term := Term * Factor
6 Term := Term / Factor
7 Term := Factor
8 Factor := **id**
9 Factor := **int**

S
by 1 ⇒ Exp
by 2 ⇒ Exp + Term
by 5 ⇒ Exp + Term * Factor
by 8 ⇒ Exp + Term * $\mathbf{id}_x$
by 7 ⇒ Exp + Factor * $\mathbf{id}_x$
by 9 ⇒ Exp + $\mathbf{int}_3$ * $\mathbf{id}_x$
by 4 ⇒ Term + $\mathbf{int}_3$ * $\mathbf{id}_x$
by 7 ⇒ Factor + $\mathbf{int}_3$ * $\mathbf{id}_x$
by 9 ⇒ $\mathbf{int}_2$ + $\mathbf{int}_3$ * $\mathbf{id}_x$

A Rightmost Derivation In Reverse

int$_2$ + **int**$_3$ * **id**$_x$

Factor + **int**$_3$ * **id**$_x$

Term + **int** * **id**

Exp +

Lets keep track of where we are in the input.

Exp + Factor * **id**$_x$

Exp + Term * **id**$_x$

Exp + Term * Factor

Exp + Term

Exp

S

A Rightmost Derivation In Reverse

$\mathtt{int}_2$ + $\mathtt{int}_3$ * $\mathtt{id}_x$

Factor + $\mathtt{int}_3$ * $\mathtt{id}_x$

Term + $\mathtt{int}_3$ * $\mathtt{id}_x$

Exp + $\mathtt{int}_3$ * $\mathtt{id}_x$

Exp + Factor * $\mathtt{id}_x$

Exp + Term * $\mathtt{id}_x$

Exp + Term * Factor

Exp + Term

Exp

S

$\mathtt{int}_2$ ● + $\mathtt{int}_3$ * $\mathtt{id}_x$

Factor ● + $\mathtt{int}_3$ * $\mathtt{id}_x$

Term ● + $\mathtt{int}_3$ * $\mathtt{id}_x$

Exp + $\mathtt{int}_3$ ● * $\mathtt{id}_x$

Exp + Factor ● * $\mathtt{id}_x$

Exp + Term * $\mathtt{id}_x$ ●

Exp + Term * Factor ●

Exp + Term ●

Exp ●

S ●

A Rightmost Derivation In Reverse

$\mathtt{int}_2$ + $\mathtt{int}_3$ * $\mathtt{id}_x$

Factor + $\mathtt{int}_3$ * $\mathtt{id}_x$

Term + $\mathtt{int}_3$ * $\mathtt{id}_x$

Exp + $\mathtt{int}_3$ * $\mathtt{id}_x$

Exp + Factor * $\mathtt{id}_x$

Exp + Term * id

Exp + Term *

Exp + Term

Exp

S

$\mathtt{int}_2$ ● + $\mathtt{int}_3$ * $\mathtt{id}_x$

Factor ● + $\mathtt{int}_3$ * $\mathtt{id}_x$

Term ● + $\mathtt{int}_3$ * $\mathtt{id}_x$

Exp + $\mathtt{int}_3$ ● * $\mathtt{id}_x$

Exp + Factor ● * $\mathtt{id}_x$

Exp + Term * $\mathtt{id}_x$ ●

Factor ●

Exp ●

S ●

Lets format this differently,
<prefix of sentential form> input

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $
int$_2$	+ **int**$_3$ * **id**$_x$ $
Factor	+ **int**$_3$ * **id**$_x$ $
Term	+ **int**$_3$ * **id**$_x$ $
Exp	+ **int**$_3$ * **id**$_x$ $
Exp +	**int**$_3$ * **id**$_x$ $
Exp + **int**$_3$	* **id**$_x$ $
Exp + Factor	* **id**$_x$ $
Exp + Term	* **id**$_x$ $
Exp + Term *	**id**$_x$ $
Exp + Term * **id**$_x$	$
Exp + Term * Factor	$
Exp + Term	$
Exp	$
S	$

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $
int$_2$	+ **int**$_3$ * **id**$_x$ $
Factor	+ **int**$_3$ * **id**$_x$ $
Term	+ **int**$_3$ * **id**$_x$ $
Exp	+ **int**$_3$ * **id**$_x$ $
Exp +	**int**$_3$ * **id**$_x$ $
Exp + **int**$_3$	* **id**$_x$ $
Exp + Factor	* **id**$_x$ $
Exp + Term	* **id**$_x$ $
Exp + Term *	**id**$_x$ $
Exp + Term * **id**$_x$	$
S	$

LR-Parser either:

1. shifts a terminal or
2. reduces by a production.

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	
Factor	+ **int**$_3$ * **id**$_x$ $	
Term	+ **int**$_3$ * **id**$_x$ $	
Exp	+ **int**$_3$ * **id**$_x$ $	
Exp +	**int**$_3$ * **id**$_x$ $	
Exp + **int**$_3$	* **id**$_x$ $	
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor		
Term		
Exp		
Exp +		
Exp + **int**$_3$	* **id**$_x$ $	
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

When we reduce by a production: A → β,
β is on right side of sentential form.

E.g., here β is 'int' and production is F → int

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	
Exp	+ **int**$_3$ * **id**$_x$ $	
Exp +	**int**$_3$ * **id**$_x$ $	
Exp + **int**$_3$	* **id**$_x$ $	
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ $	
Exp +	**int**$_3$ * **id**$_x$ $	
Exp + **int**$_3$	* **id**$_x$ $	
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	
Exp + **int**$_3$	* **id**$_x$ $	
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	shift 3
Exp + **int**$_3$	* **id**$_x$ $	
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	shift 3
Exp + **int**$_3$	* **id**$_x$ $	reduce by F → int
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	shift 3
Exp + **int**$_3$	* **id**$_x$ $	reduce by F → int
Exp + Factor	* **id**$_x$ $	reduce by F → T
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	shift 3
Exp + **int**$_3$	* **id**$_x$ $	reduce by F → int
Exp + Factor	* **id**$_x$ $	reduce by F → T
Exp + Term	* **id**$_x$ $	shift *
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	shift 3
Exp + **int**$_3$	* **id**$_x$ $	reduce by F → int
Exp + Factor	* **id**$_x$ $	reduce by F → T
Exp + Term	* **id**$_x$ $	shift *
Exp + Term *	**id**$_x$ $	shift x
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	shift 3
Exp + **int**$_3$	* **id**$_x$ $	reduce by F → int
Exp + Factor	* **id**$_x$ $	reduce by F → T
Exp + Term	* **id**$_x$ $	shift *
Exp + Term *	**id**$_x$ $	shift x
Exp + Term * **id**$_x$	$	reduce by F → id
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	shift 3
Exp + **int**$_3$	* **id**$_x$ $	reduce by F → int
Exp + Factor	* **id**$_x$ $	reduce by F → T
Exp + Term	* **id**$_x$ $	shift *
Exp + Term *	**id**$_x$ $	shift x
Exp + Term * **id**$_x$	$	reduce by F → id
Exp + Term * Factor	$	reduce by T → T * F
Exp + Term	$	
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	shift 3
Exp + **int**$_3$	* **id**$_x$ $	reduce by F → int
Exp + Factor	* **id**$_x$ $	reduce by F → T
Exp + Term	* **id**$_x$ $	shift *
Exp + Term *	**id**$_x$ $	shift x
Exp + Term * **id**$_x$	$	reduce by F → id
Exp + Term * Factor	$	reduce by T → T * F
Exp + Term	$	reduce by E → E + T
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	shift 3
Exp + **int**$_3$	* **id**$_x$ $	reduce by F → int
Exp + Factor	* **id**$_x$ $	reduce by F → T
Exp + Term	* **id**$_x$ $	shift *
Exp + Term *	**id**$_x$ $	shift x
Exp + Term * **id**$_x$	$	reduce by F → id
Exp + Term * Factor	$	reduce by T → T * F
Exp + Term	$	reduce by E → E + T
Exp	$	reduce by S → E
S	$	

A Rightmost Derivation In Reverse

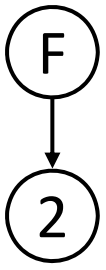
	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	shift 3
Exp + **int**$_3$	* **id**$_x$ $	reduce by F → int
Exp + Factor	* **id**$_x$ $	reduce by F → T
Exp + Term	* **id**$_x$ $	shift *
Exp + Term *	**id**$_x$ $	shift x
Exp + Term * **id**$_x$	$	reduce by F → id
Exp + Term * Factor	$	reduce by T → T * F
Exp + Term	$	reduce by E → E + T
Exp	$	reduce by S → E
S	$	accept!

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	
Factor	+ **int**$_3$ * **id**$_x$ $	
Term	+ **int**$_3$ * **id**$_x$ $	
Exp	+ **int**$_3$ * **id**$_x$ $	
Exp +	**int**$_3$ * **id**$_x$ $	
Exp + **int**$_3$	* **id**$_x$ $	
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	(2)
S	$	

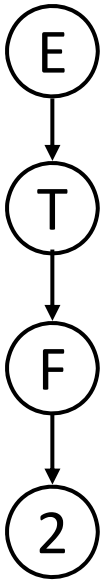
A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	
Term	+ **int**$_3$ * **id**$_x$ $	
Exp	+ **int**$_3$ * **id**$_x$ $	
Exp +	**int**$_3$ * **id**$_x$ $	
Exp + **int**$_3$	* **id**$_x$ $	
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	



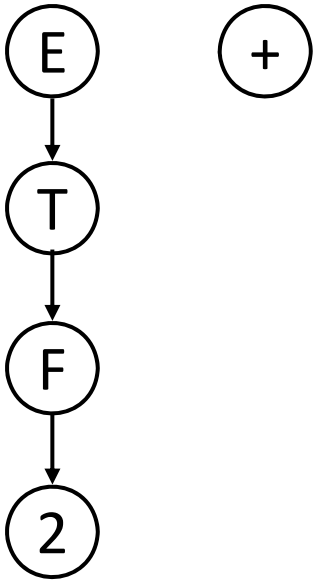
A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ $	
Exp +	**int**$_3$ * **id**$_x$ $	
Exp + **int**$_3$	* **id**$_x$ $	
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	



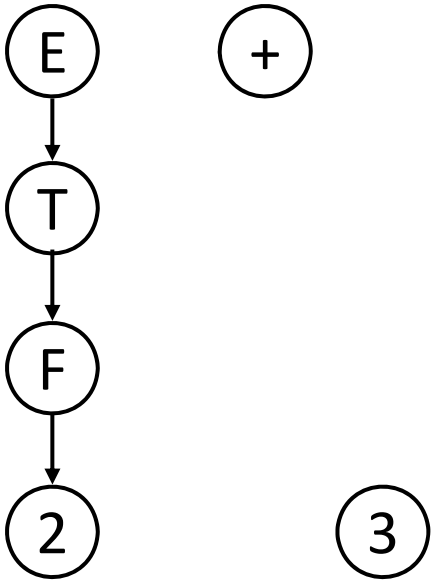
A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	
Exp + **int**$_3$	* **id**$_x$ $	
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	



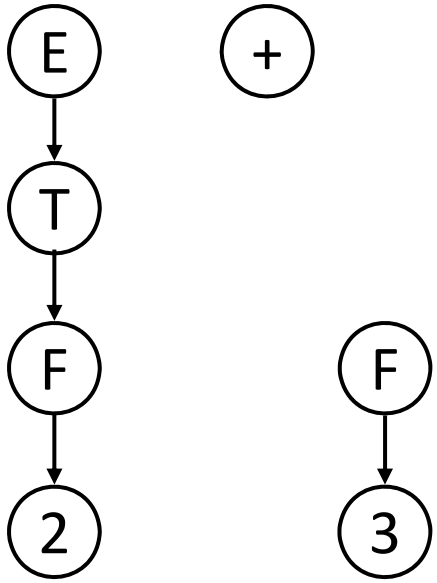
A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	shift 3
Exp + **int**$_3$	* **id**$_x$ $	
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	



A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	shift 3
Exp + **int**$_3$	* **id**$_x$ $	reduce by F → int
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	



Handles

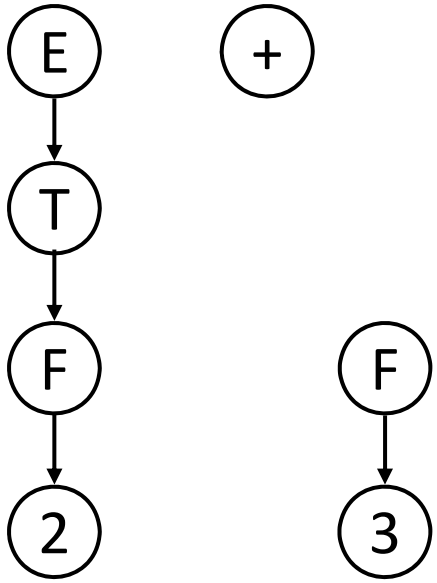
- LR parsing is handle pruning
- LR parsing finds a rightmost derivation (in reverse)
- A handle in γ, a right-hand sentential form, is
  - a position in γ matching β
  - a production $A \rightarrow \beta$

$$S \rightarrow^* \alpha A w \rightarrow \alpha \beta w$$

- if a grammar is unambiguous, then every γ has exactly 1 handle

A Rightmost Derivation In Reverse

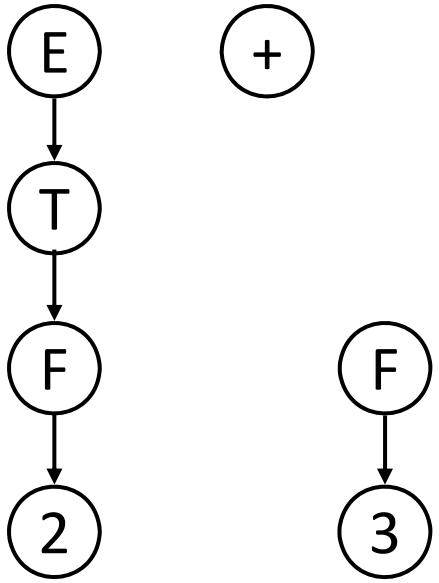
	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	shift 3
Exp + **int**$_3$	* **id**$_x$ $	reduce by F → int
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	



A Rightmost Derivation In Reverse

Where is next handle?

int$_2$	**int**$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Term	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Exp	+ **int**$_3$ * **id**$_x$ $	reduce by T → E
Exp +	**int**$_3$ * **id**$_x$ $	shift +
Exp + **int**$_3$	* **id**$_x$ $	shift 3
Exp + Factor	* **id**$_x$ $	reduce by F → int
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	



A Rightmost … Reverse

Where is next handle?

int$_2$	
Factor	
Term	
Exp	
Exp +	
Exp + **int**$_3$	
Exp + Factor	* **id**$_x$ $
Exp + Term	* **id**$_x$ $
Exp + Term *	**id**$_x$ $
Exp + Term * **id**$_x$	$
Exp + Term * Factor	$
Exp + Term	$
Exp	$
S	$

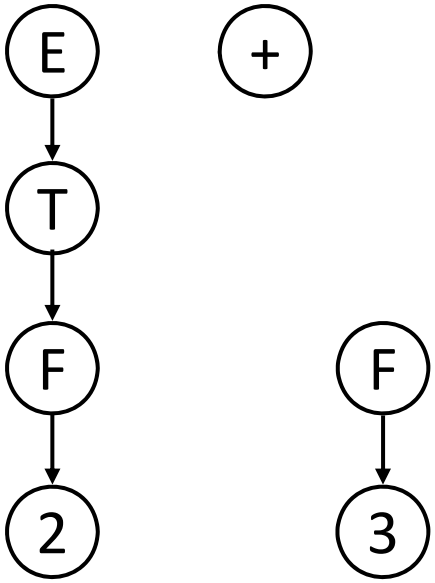
1	S	:= Exp
2	Exp	:= Exp + Term
3	Exp	:= Exp - Term
4	Exp	:= Term
5	Term	:= Term * Factor
6	Term	:= Term / Factor
7	Term	:= Factor
8	Factor	:= **id**
9	Factor	:= **int**

… by F → int

… by T → F

… by T → E

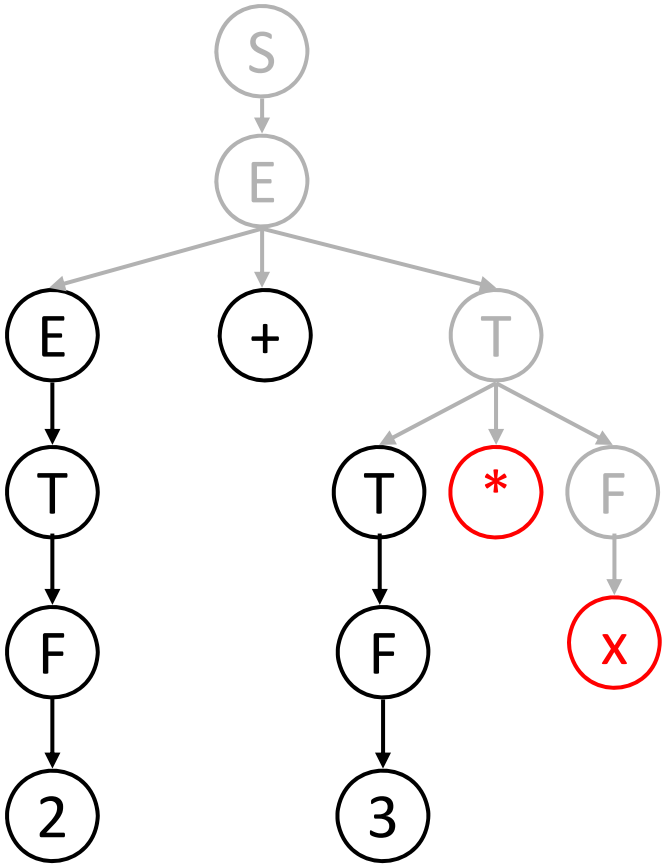
… by F → int



A Rightmost Derivation In Reverse

Where is next handle? E+F*x and T→ F

int$_2$	+ **int**$_3$ * **id**$_x$ $
Factor	+ **int**$_3$ * **id**$_x$ $
Term	+ **int**$_3$ * **id**$_x$ $
Exp	+ **int**$_3$ * **id**$_x$ $
Exp +	**int**$_3$ * **id**$_x$ $
Exp + **int**$_3$	* **id**$_x$ $
Exp + Factor	* **id**$_x$ $
Exp + Term	* **id**$_x$ $
Exp + Term *	**id**$_x$ $
Exp + Term * **id**$_x$	$
Exp + Term * Factor	$
Exp + Term	$
Exp	$
S	$



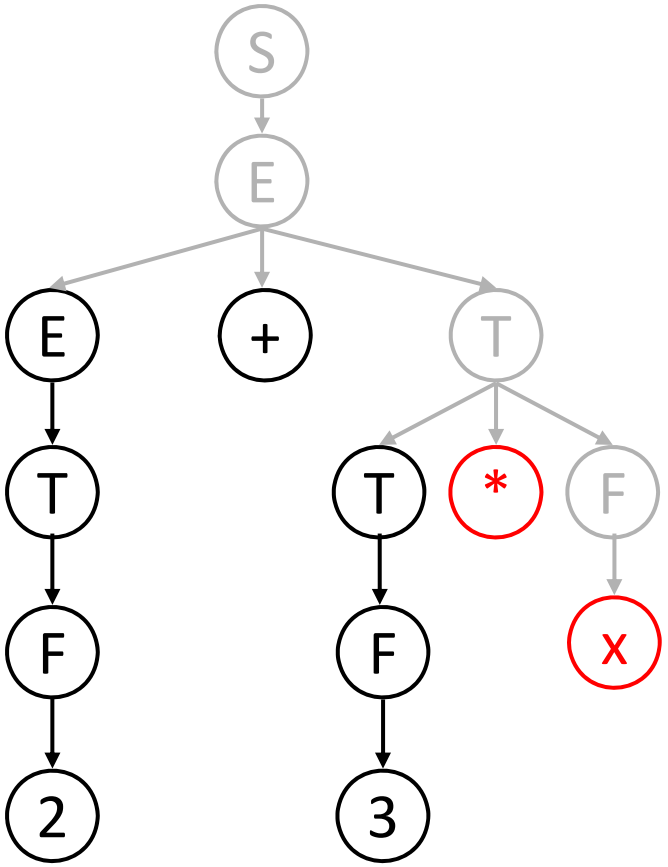
Handle Pruning

- LR parsing consists of
  - shifting til there is a handle on the top of the stack
  - reducing handle
- Key is handle is always on top of stack, i.e., if β is a handle with $A \rightarrow \beta$, then β can be found on top of stack.

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $
int$_2$	+ **int**$_3$ * **id**$_x$ $
Factor	+ **int**$_3$ * **id**$_x$ $
Term	+ **int**$_3$ * **id**$_x$ $
Exp	+ **int**$_3$ * **id**$_x$ $
Exp +	**int**$_3$ * **id**$_x$ $
Exp + **int**$_3$	* **id**$_x$ $
Exp + Factor	* **id**$_x$ $
Exp + Term	* **id**$_x$ $
Exp + Term *	**id**$_x$ $
Exp + Term * **id**$_x$	$
Exp + Term * Factor	$
Exp + Term	$
Exp	$
S	$

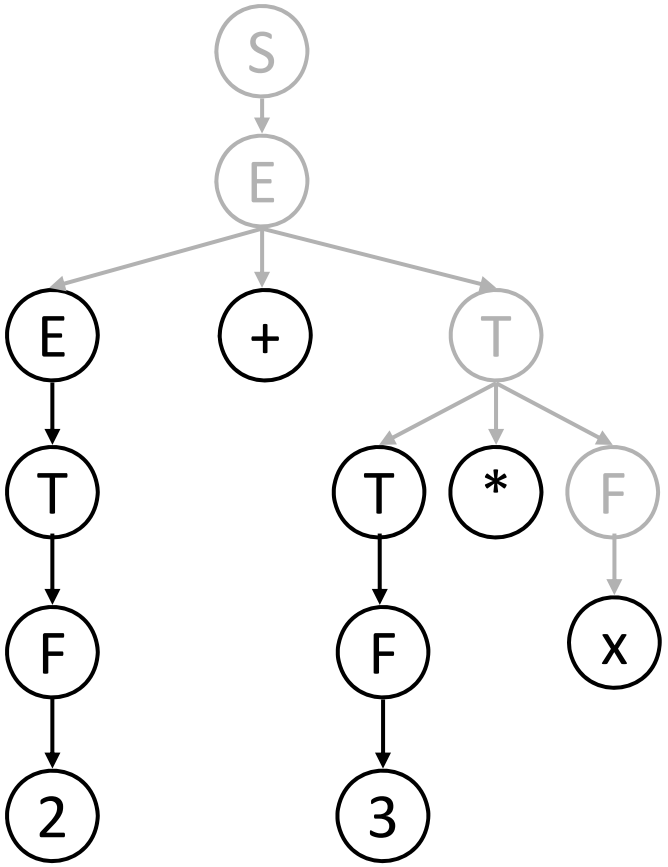
top of stack does not have a handle, so must shift.



A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $
int$_2$	+ **int**$_3$ * **id**$_x$ $
Factor	+ **int**$_3$ * **id**$_x$ $
Term	+ **int**$_3$ * **id**$_x$ $
Exp	+ **int**$_3$ * **id**$_x$ $
Exp +	**int**$_3$ * **id**$_x$ $
Exp + **int**$_3$	* **id**$_x$ $
Exp + Factor	* **id**$_x$ $
Exp + Term	* **id**$_x$ $
Exp + Term *	**id**$_x$ $
Exp + Term * **id**$_x$	$
Exp + Term * Factor	$
Exp + Term	$
Exp	$
S	$

Now, x is a handle.

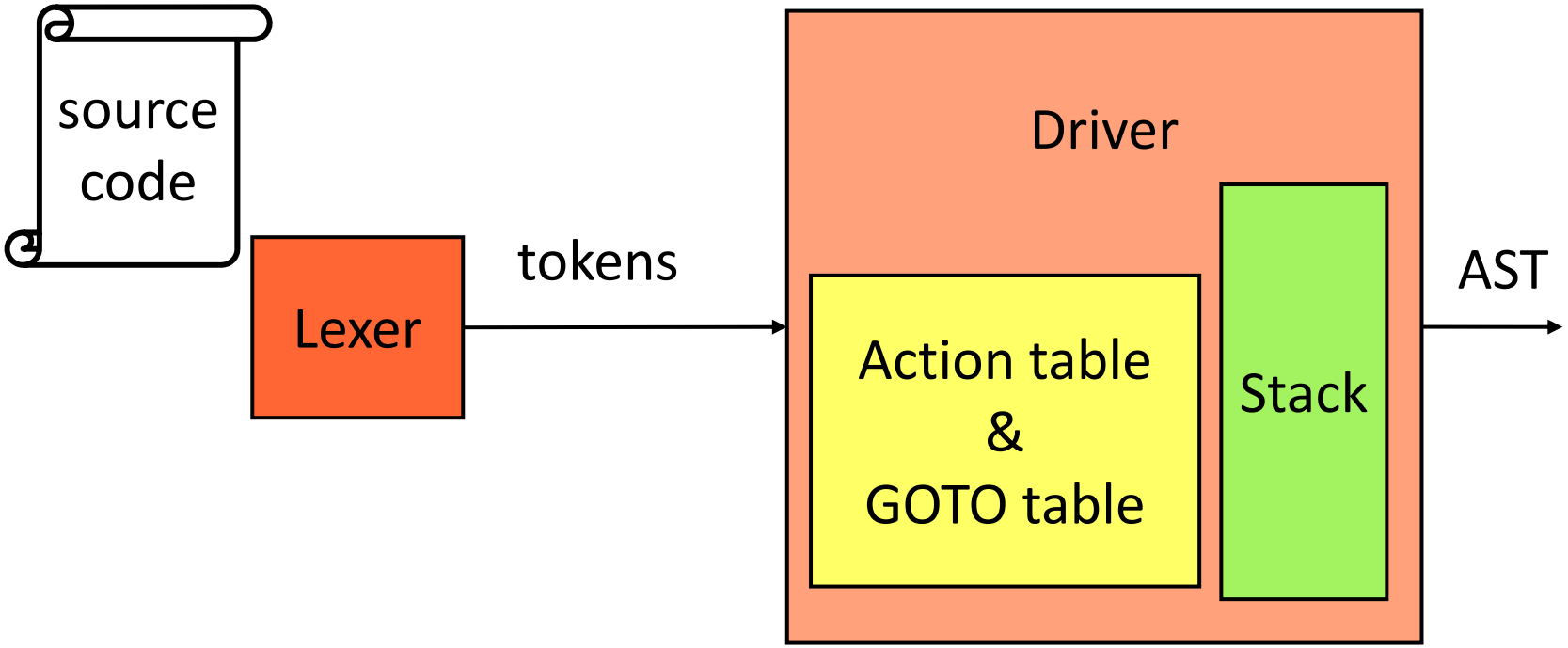


A Shift-Reduce Parser

- Stack holds the viable prefixes.
- input stream holds remaining source
- Four actions:
  - shift: push token from input stream onto stack
  - reduce: right-end of a handle (β of $A \rightarrow \beta$) is at top of stack, pop handle (β), push A
  - accept: success
  - error: syntax error discovered

Key is recognizing handles efficiently

Table-driven LR(k) parsers



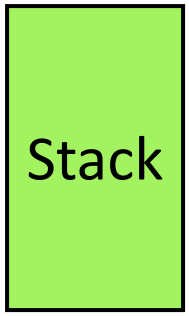
Push down automata:
FSM with stack

Parser Loop

Driver

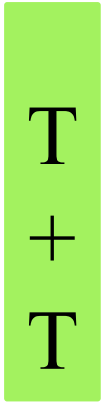
- Same code regardless of grammar
  - only tables change
- (Very) General Algorithm:
  - Based on table contents, top of stack, and current input character either
    - **shift**: pushes onto stack, reads next token
    - **reduce**: manipulate stack to simplify representation of already scanned input
    - **accept**: successfully scanned entire input
    - **error**: input not in language

Stack



- Represents the scanned input
- Contents?
  - Reduced nonterminals not enough
  - Must store previously seen *states*
    - the context of the current position
  - In fact, nonterminals unnecessary
    - include for readability

$$x + y\bullet + z$$



Parser Tables

Action table
&
GOTO table

Action table

- given state *s* and **terminal** *a* tells parser loop what action (shift, reduce, accept, reject) to perform

Goto table

- used when performing reduction; given a state *s* and **nonterminal** *X* says what state to transition to

Parser Tables

Action table
&
GOTO table

s*N* push state *N* onto stack
r*R* reduce by rule *R*
g*N* goto state *N*
a accept
error

	action			goto	
state	*ident*	+	$	E	T
0	**s3**			**g1**	**g2**
1			**a**		
2		**s4**	**r2**		
3		**r3**	**r3**		
4	**s3**			**g5**	**g2**
5			**r1**		

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Parser Loop Revisited

Driver

```
while(true)
   s = state on top of stack
   a = current input token
   if(action[s][a] == sN)                    shift
      push N
      read next input token
   else if(action[s][a] == rR)               reduce
      pop rhs of rule R from stack
      X = lhs of rule R
      N = state on top of stack
      push goto[N][X]
   else if(action[s][a] == a)                accept
      return success
   else                                      error
      return failure
```

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

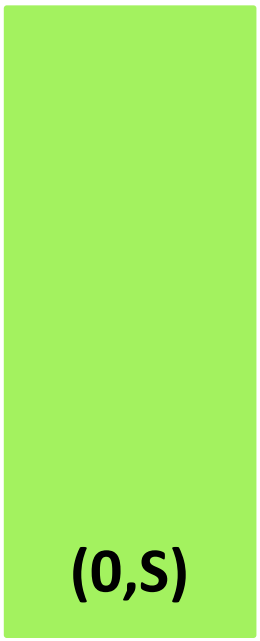
2 E → T

3 T → *identifier*

Current input token = **x**

State on top of the stack = **0**

x + y$



Stack

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

Current input token = **+**

State on top of the stack = **3**

x + y$

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

(3,x)
(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **+**

State on top of the stack = **3**

x + y$

(3,x)
(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

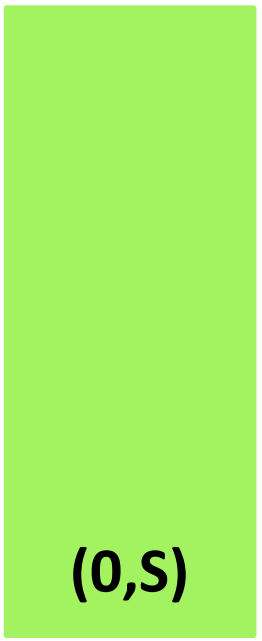
1 E → T + E

2 E → T

3 T → *identifier*

Current input token = +

State on top of the stack = 3



x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

Current input token = **+**

State on top of the stack = **0**

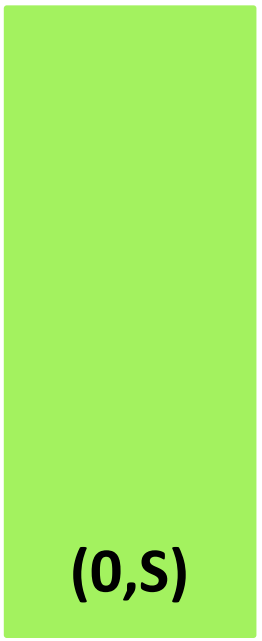
x + y$

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*



Example

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **+**

State on top of the stack = **2**

x + y$

(2,T)
(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **+**

State on top of the stack = **2**

x + y$

(2,T)
(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

Current input token = **y**
State on top of the stack = **4**

x + y$

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

(4,+)
(2,T)
(0,S)

Example

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

Current input token = **y**
State on top of the stack = **4**

x + y$

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

(4,+)
(2,T)
(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **3**

x + y$

(3,y)
(4,+)
(2,T)
(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

Current input token = **$**

State on top of the stack = **3**

x + y$

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

(4,+)
(2,T)
(0,S)

(?,T)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

Current input token = **$**
State on top of the stack = **2**

x + y$

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

(2,T)
(4,+)
(2,T)
(0,S)

Example

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **2**

x + y$

(2,T)
(4,+)
(2,T)
(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

Current input token = **$**

State on top of the stack = **2**

x + y$

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

(?,E)

(4,+)
(2,T)
(0,S)

Example

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

Current input token = **$**
State on top of the stack = **5**

x + y$

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

(5,E)
(4,+)
(2,T)
(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

Current input token = **$**
State on top of the stack = **5**

x + y$

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

(5,E)
(4,+)
(2,T)
(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

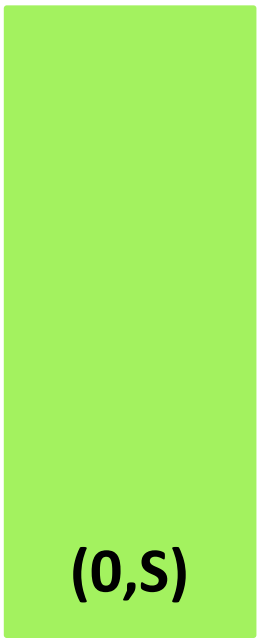
1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **5**



x + y$

Example

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

Current input token = **$**

State on top of the stack = **1**

x + y$

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

(1,E)

(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

Accept!

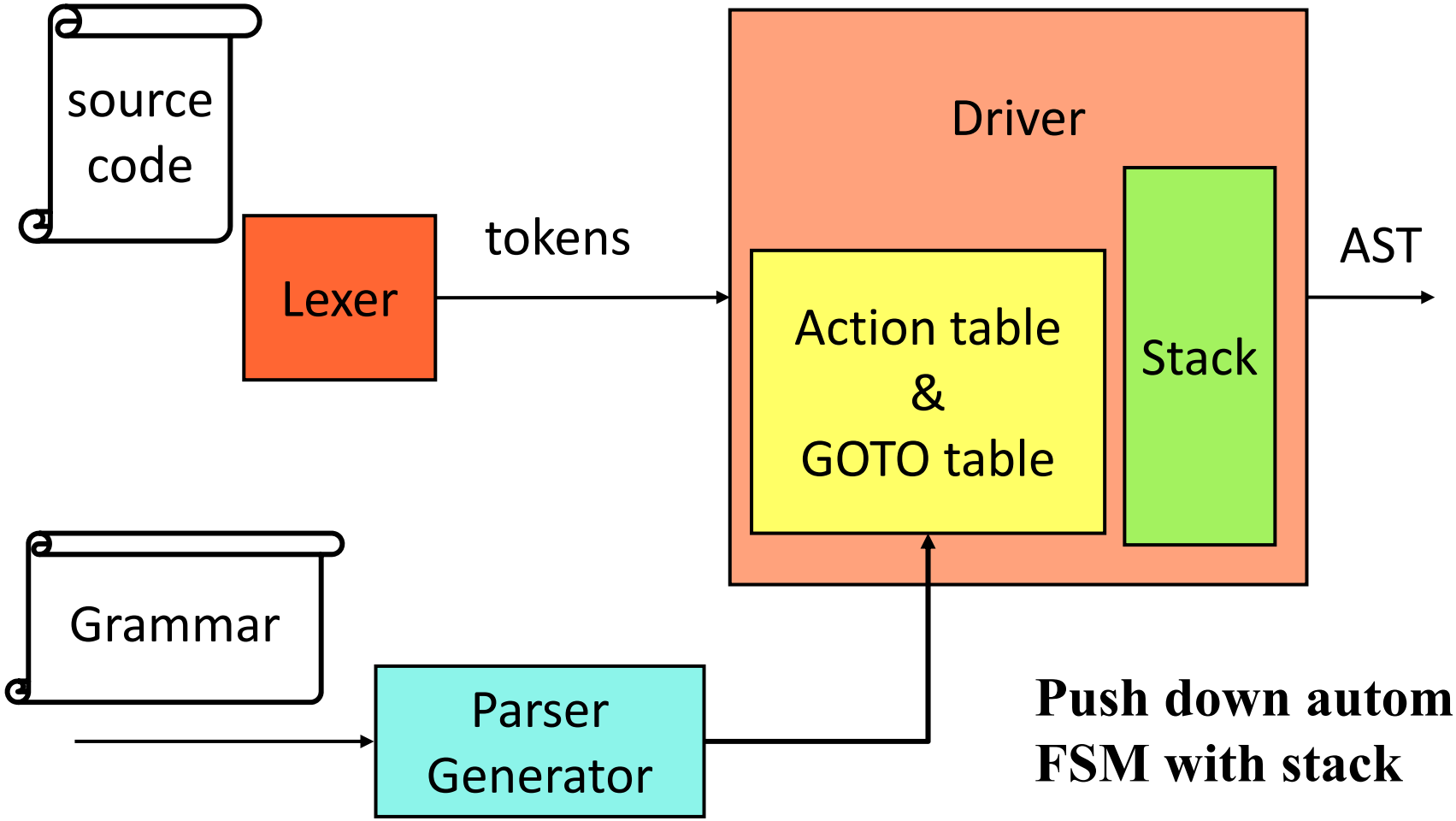
0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Current input token = **$**
State on top of the stack = **1**

x + y$

(1,E)
(0,S)

Table-driven LR(k) parsers



Push down automata: FSM with stack

The parser generator

Parser Generator

- Finds handles
- Creates the action and GOTO tables.
- Creates the states
  - Each state indicates how much of a handle we have seen
  - each state is a set of *items*

Items

- Items are used to identify handles.
- LR(k) items have the form:

  [production-with-dot, lookahead]
- For example, A $\rightarrow$ a X b has 4 LR(0) items
  - [A $\rightarrow$ • a X b]
  - [A $\rightarrow$ a • X b]
  - [A $\rightarrow$ a X • b]
  - [A $\rightarrow$ a X b •]

The • indicates how much of the handle we have recognized.

What LR(0) Items Mean

- $[X \rightarrow \bullet\ \alpha\ \beta\ \gamma]$
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$
- $[X \rightarrow \alpha\ \bullet\ \beta\ \gamma]$
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$ and we have already recognized α
- $[X \rightarrow \alpha\ \beta\ \bullet\ \gamma]$
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$ and we have already recognized $\alpha\ \beta$
- $[X \rightarrow \alpha\ \beta\ \gamma\ \bullet]$
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$ and we can reduce to X

Generating the States

- Start with start production.
- In this case, "S → E$"

$S \rightarrow \bullet E\$$

0 $S \rightarrow E\$$
1 $E \rightarrow T + E$
2 $E \rightarrow T$
3 $T \rightarrow \textit{identifier}$

- Each state is consistent with what we have already shifted from the input and what is possible to reduce. So, what other items should be in this state?

Completing a state

- For each item in a state, add in all other consistent items.

$S \rightarrow \bullet E\$$
$E \rightarrow \bullet T + E$
$E \rightarrow \bullet T$
$T \rightarrow \bullet identifier$

0 $S \rightarrow E\$$
1 $E \rightarrow T + E$
2 $E \rightarrow T$
3 $T \rightarrow identifier$

- This is called, taking the closure of the state.

Closure*

```
closure(state)
   repeat
      foreach item A → a•Xb in state
         foreach production X → w
            state.add(X → •w)
   until state does not change
   return state
```

Intuitively:

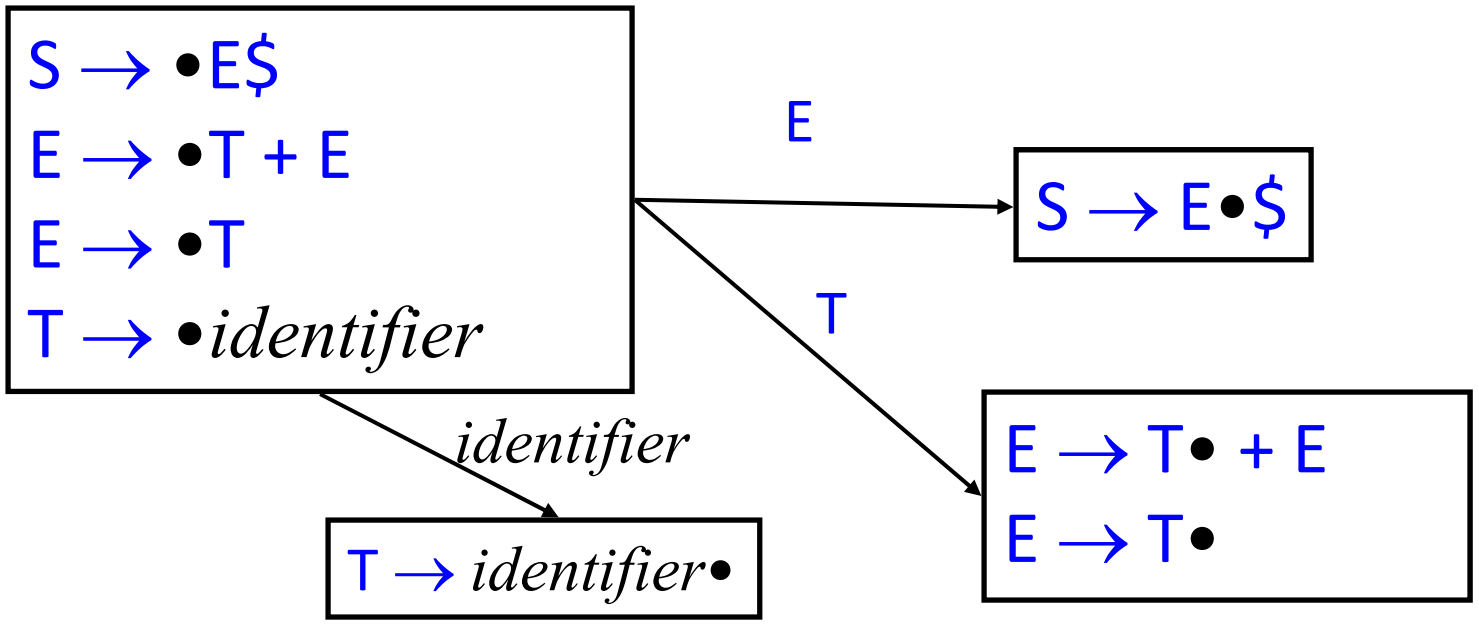
Given a set of items, add all production rules that could produce the nonterminal(s) at the current position in each item

*: for LR(0) items

What about the other states?

- How do we decide what the other states are?
- How do we decide what the transitions between states are?

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*



Next(state, sym)

- Next function determines what state to goto based on current state and symbol being recognized.
- For Non-terminal, this is used to determine the GOTO table.
- For terminal, this is used to determine the shift action.

Constructing states

```
initial_state = closure({start production})
state_set.add(initial_state)
state_queue.push(initial_state)

while(!state_queue.empty())
   s = state_queue.pop()
   foreach item A → a•Xb in s
      n = closure(next(s, X))
      if(!state_set.contains(n))
         state_set.add(n)
         state_queue.push(n)
```

A state is a set of LR(0) items

get "next" state

Closure*

closure({S → •E$}) =

S → •E$

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

*: for LR(0) items

Closure*

closure({S → •E$}) =

S → •E$
E → •T + E
E → •T
T → •*identifier*

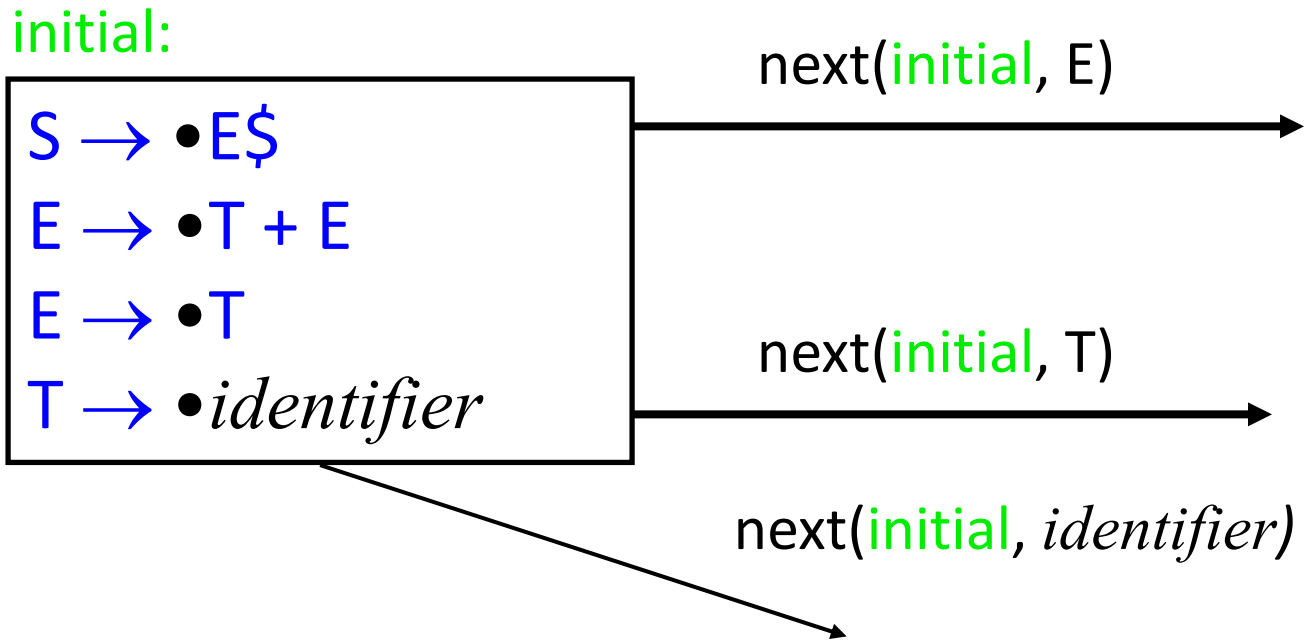
0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

*: for LR(0) items

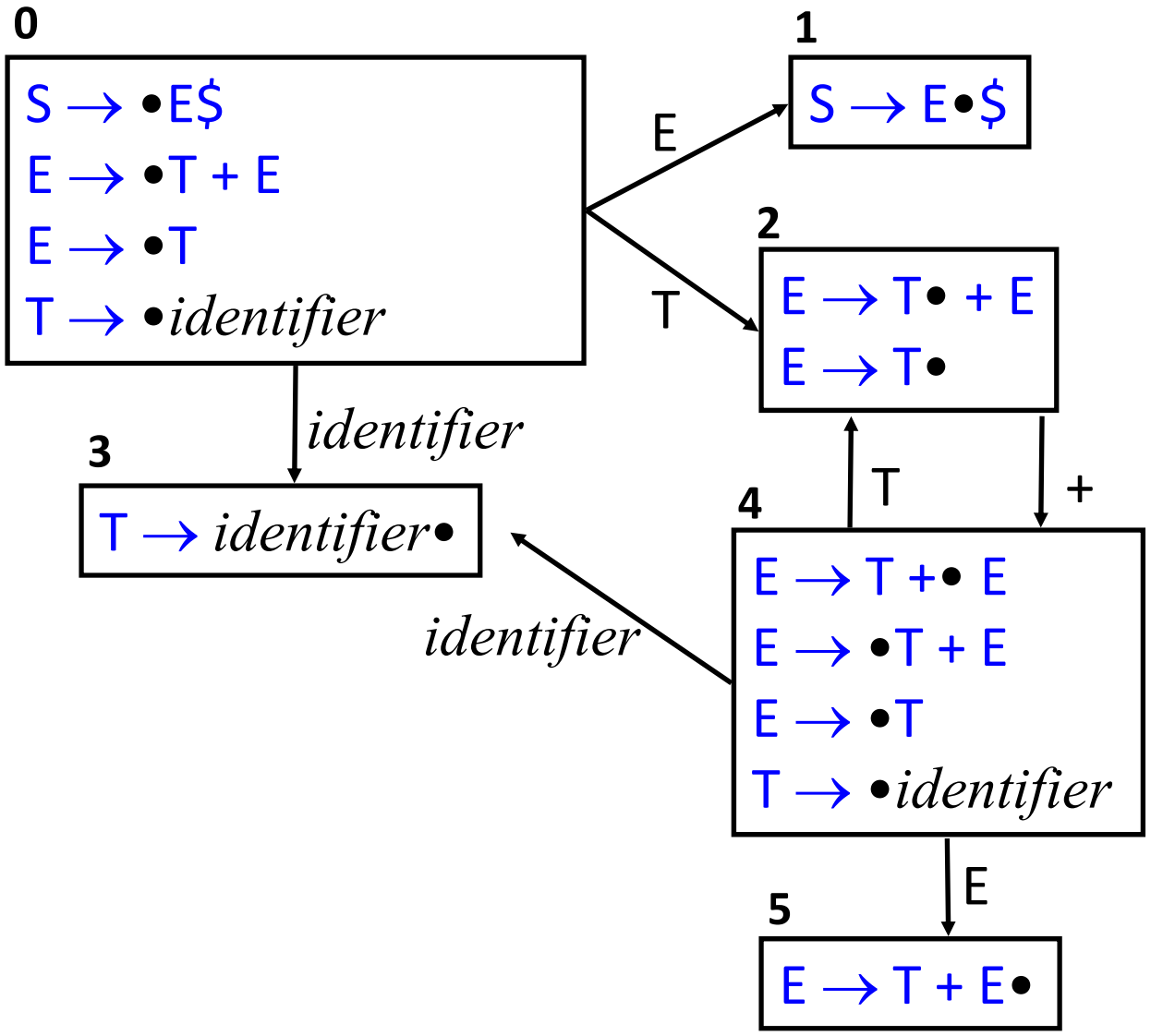
Next

```
next(state, X)
    ret = empty
    foreach item A → a•Xb in state
        ret.add(A → aX•b)
    return ret
```

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*



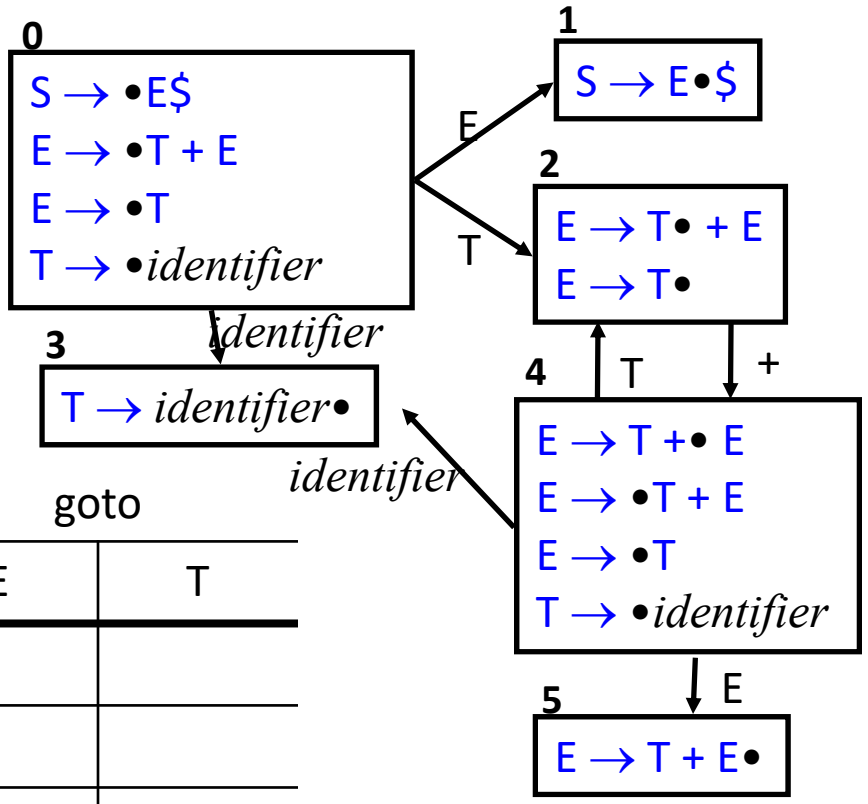
Example



0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Parse Tables for LR(0) parser

What can we fill out?



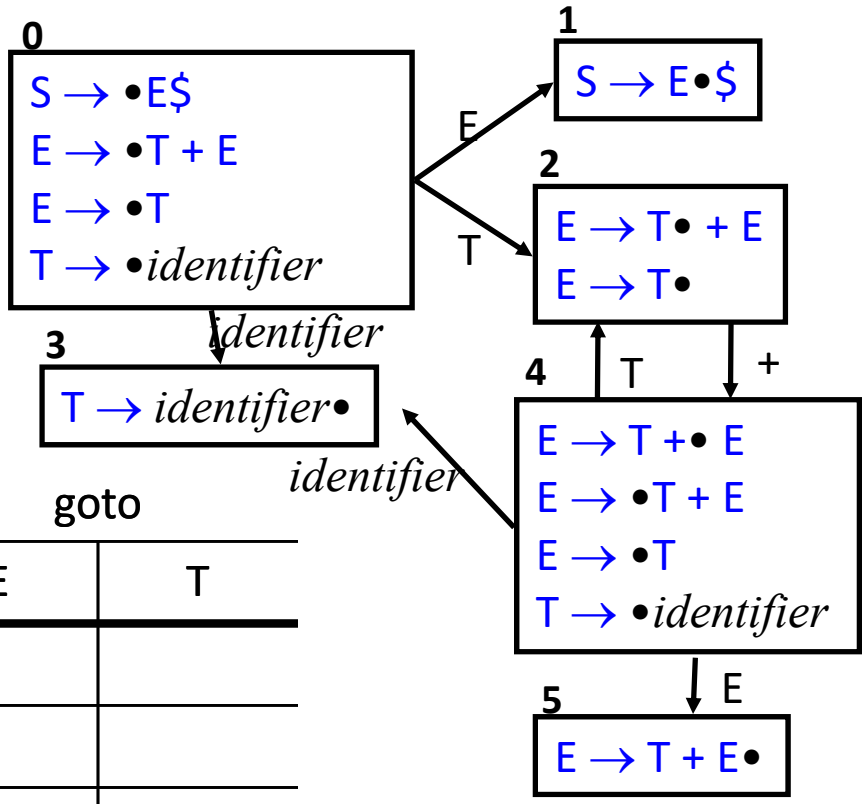
	action			goto	
state	*ident*	+	$	E	T
0					
1					
2					
3					
4					
5					

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Parse Tables for LR(0) parser

shift

transition on terminal



	action			goto	
state	*ident*	+	$	E	T
0	s3				
1					
2		s4			
3					
4	s3				
5					

0 S → E$

1 E → T + E

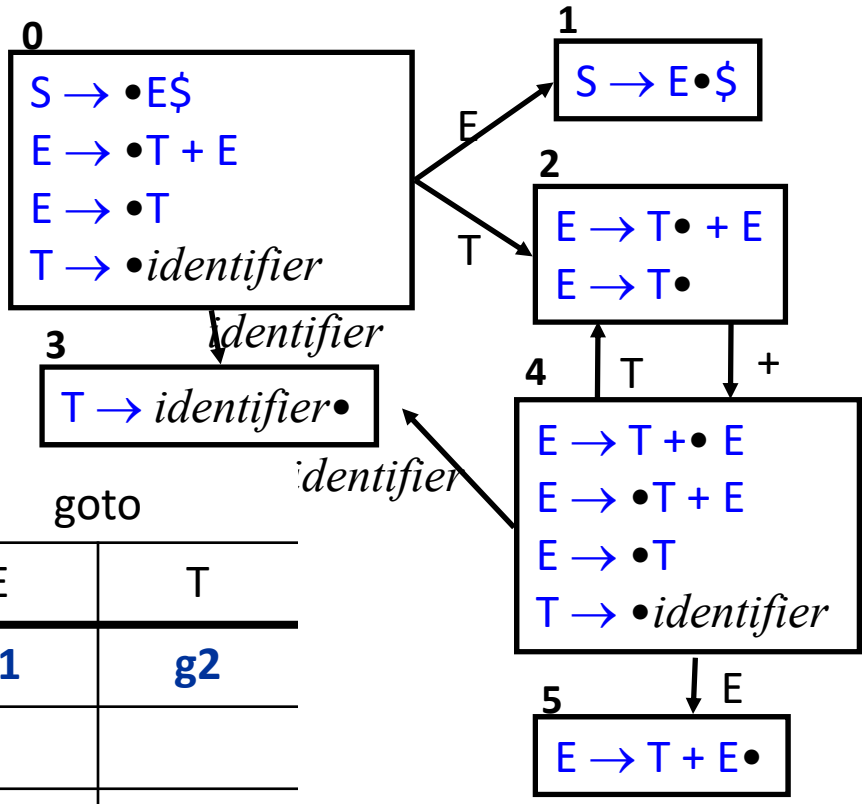
2 E → T

3 T → *identifier*

Parse Tables for LR(0) parser

goto

transition on nonterminal



state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1					
2		s4			
3					
4	s3			g5	g2
5					

0 S → E$

1 E → T + E

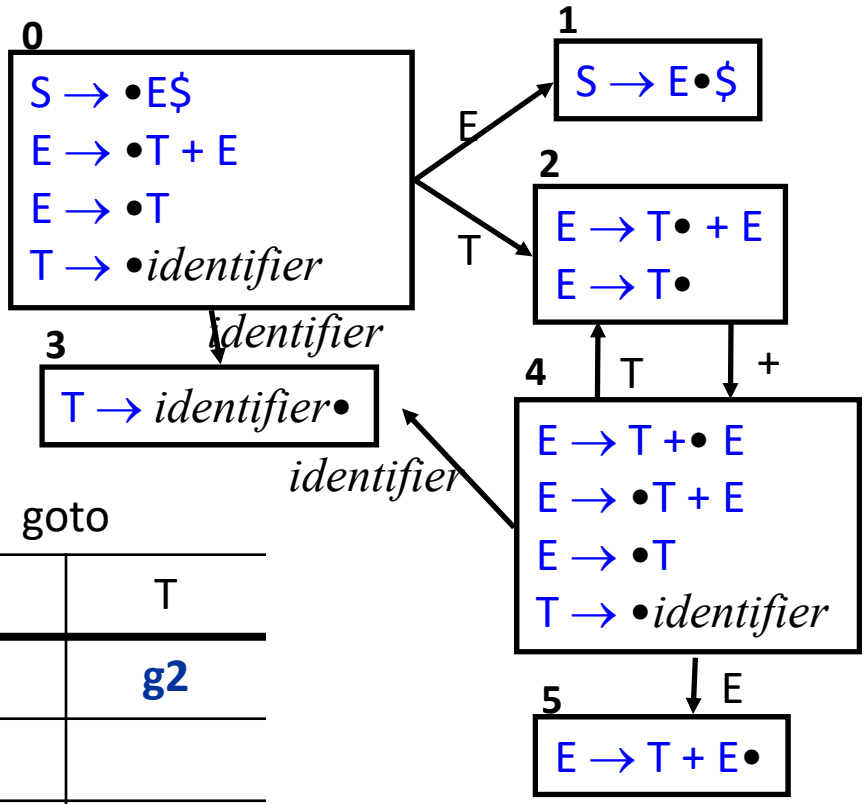
2 E → T

3 T → *identifier*

Parse Tables for LR(0) parser

accept

about to shift $



	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4			
3					
4	s3			g5	g2
5					

0 S → E$

1 E → T + E

2 E → T

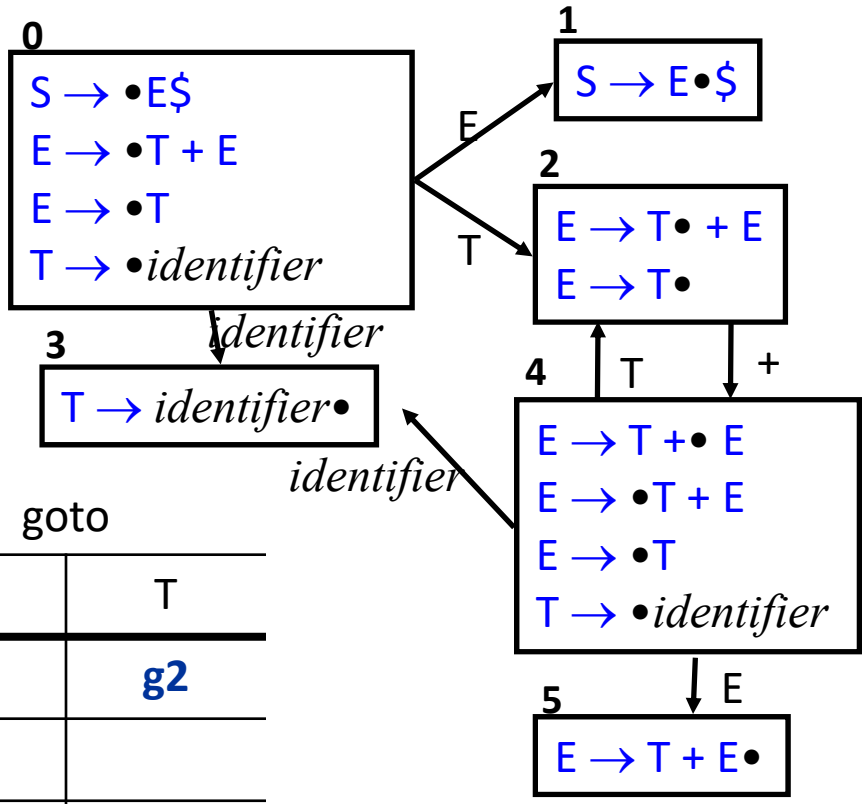
3 T → *identifier*

Parse Tables for LR(0) parser

reduce

item has dot at end

$A \rightarrow w\bullet$



state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4			
3					
4	s3			g5	g2
5					

0 S → E$

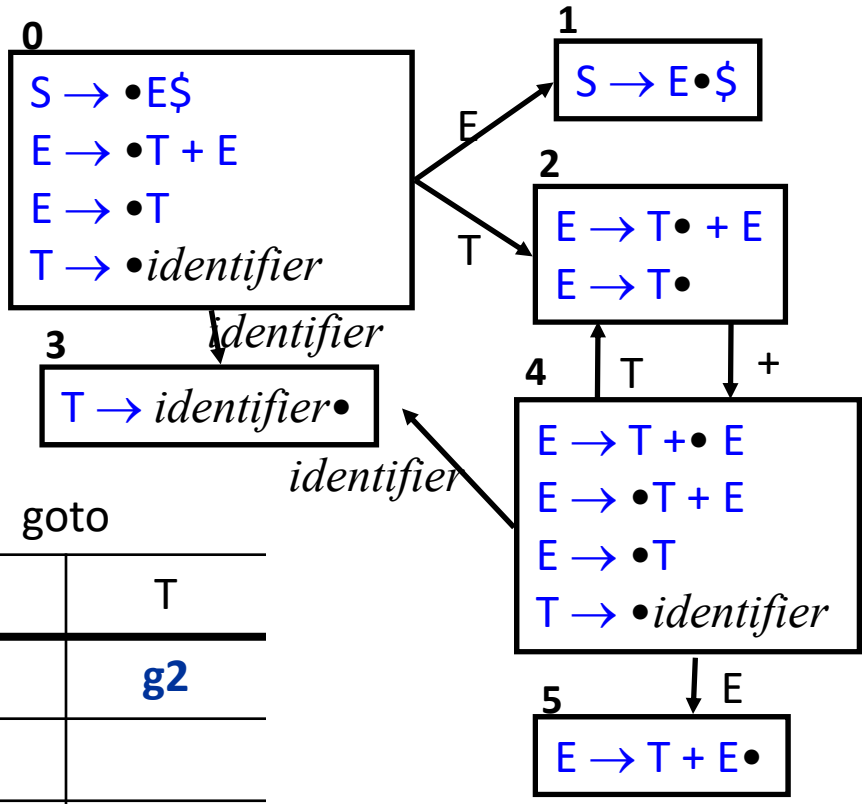
1 E → T + E

2 E → T

3 T → *identifier*

LR(0)

No lookahead
reduce state for *all* nonterminals



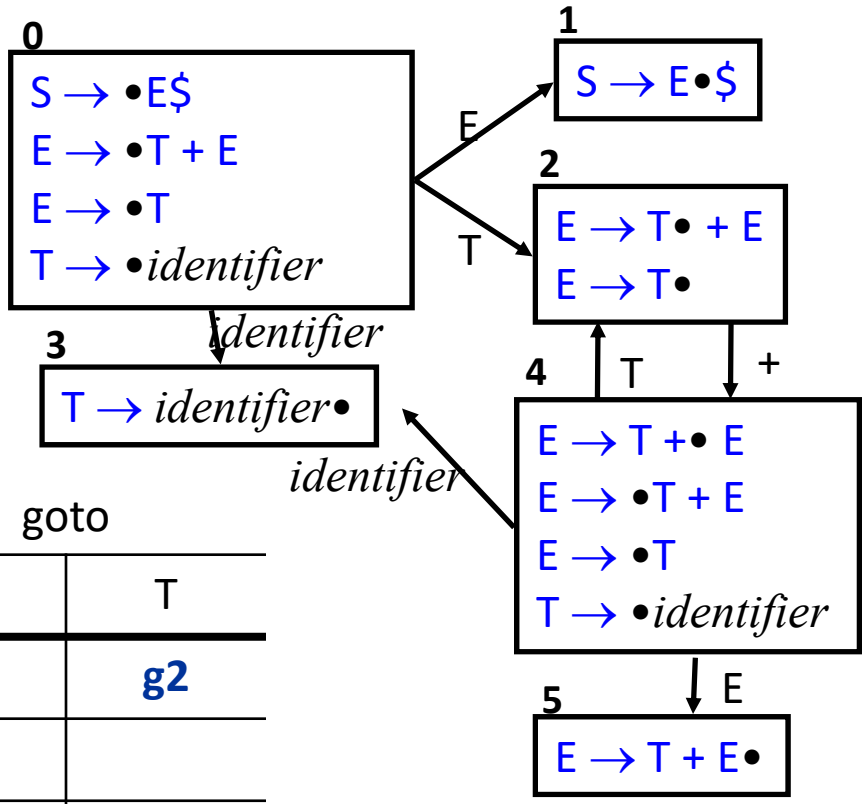
	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2	r2	r2/s4	r2		
3	r3	r3	r3		
4	s3			g5	g2
5	r1	r1	r1		

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

LR(0)

shift/reduce conflict

need to be pickier about when we reduce

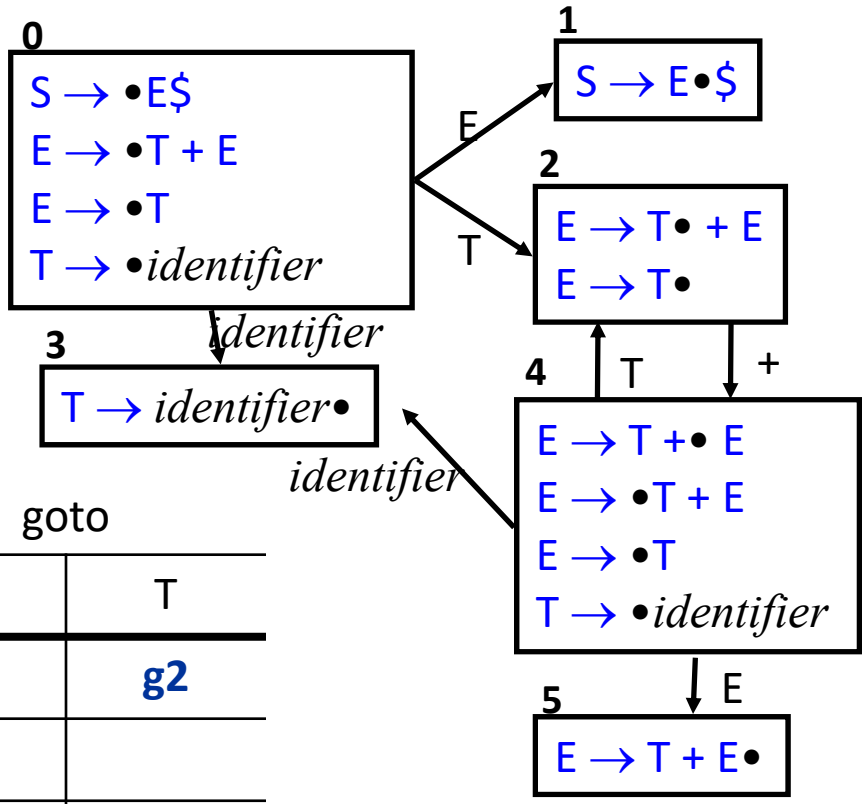


	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2	r2	r2/s4	r2		
3	r3	r3	r3		
4	s3			g5	g2
5	r1	r1	r1		

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

SLR - Simple LR

Only reduce in position (**s**,**a**) by rule R:A → *w* if **a** is in the *follow set* of A



state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4			
3					
4	s3			g5	g2
5					

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Reminder: Follow sets

follow(X)

set of terminals that can appear immediately after the nonterminal X in some sentential form

0 $S \rightarrow E\$$

1 $E \rightarrow T + E$

2 $E \rightarrow T$

3 $T \rightarrow \textit{identifier}$

I.e., $t \in$ FOLLOW(X) iff $S \Rightarrow^* \alpha X t \beta$ for some α and β

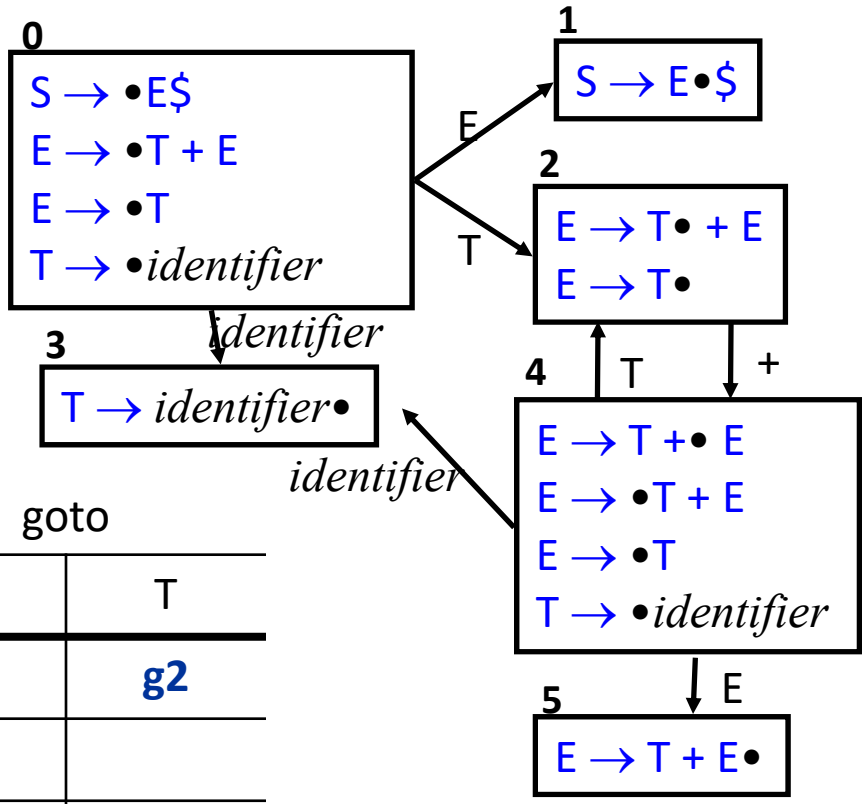
follow(E**) =** {$}

follow(T**) =** {+,$}

SLR - Reduce using follow sets

follow(E) = {$}

follow(T) = {+,$}



state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

SLR Limitations

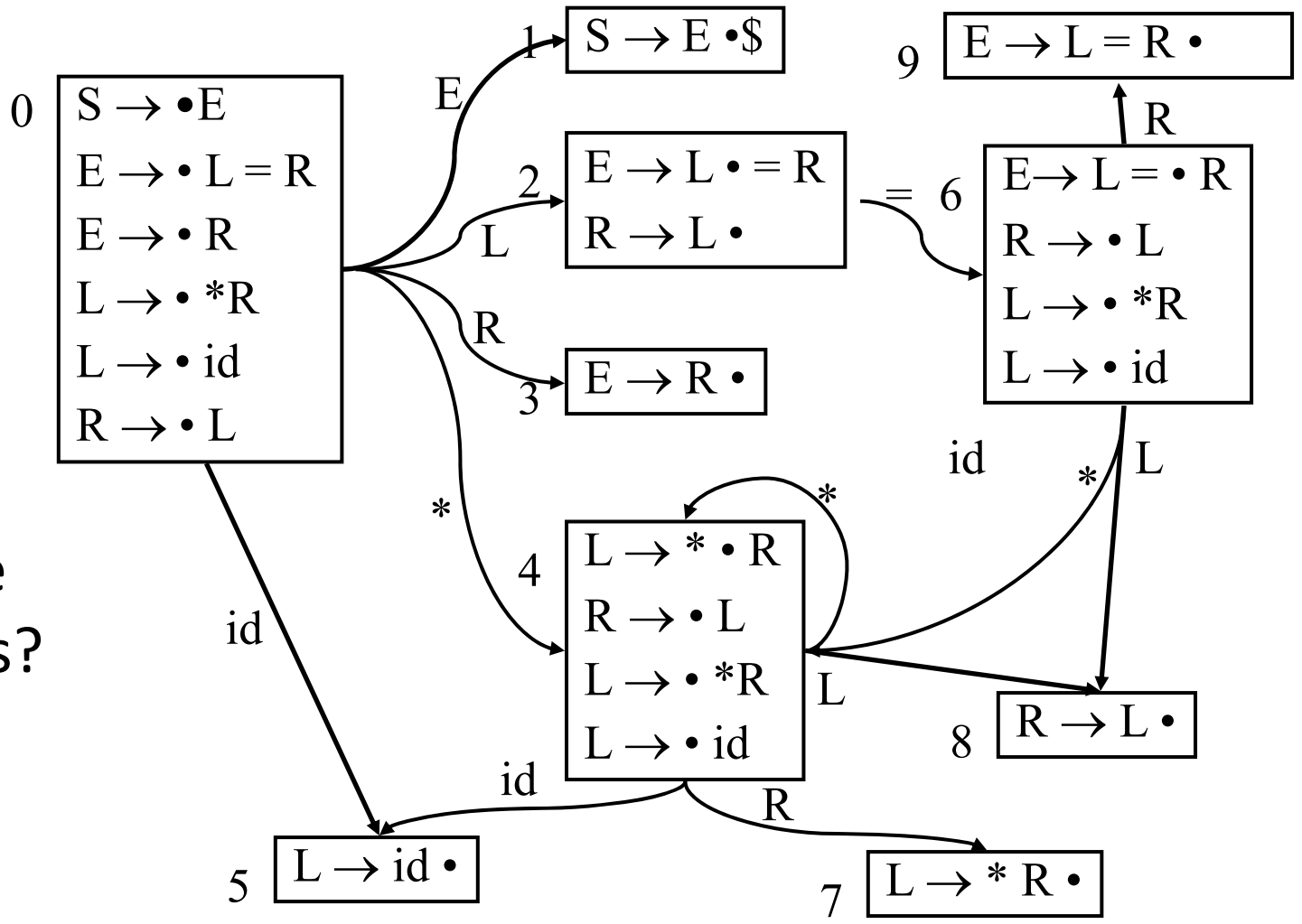
- SLR uses LR(0) item sets
- Can remove some (but not all) shift/reduce conflicts using follow set
- Consider

0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

Example

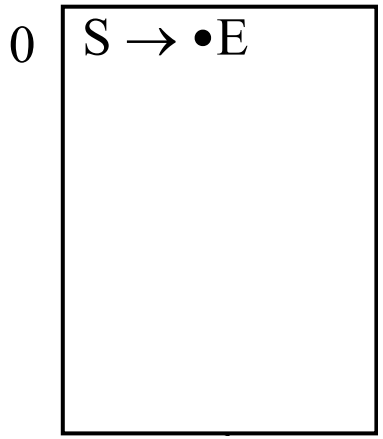
0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

What are the reduce states?



Example

0 $S \rightarrow E\$$
1 $E \rightarrow L = R$
2 $E \rightarrow R$
3 $L \rightarrow id$
4 $L \rightarrow *R$
5 $R \rightarrow L$



Example

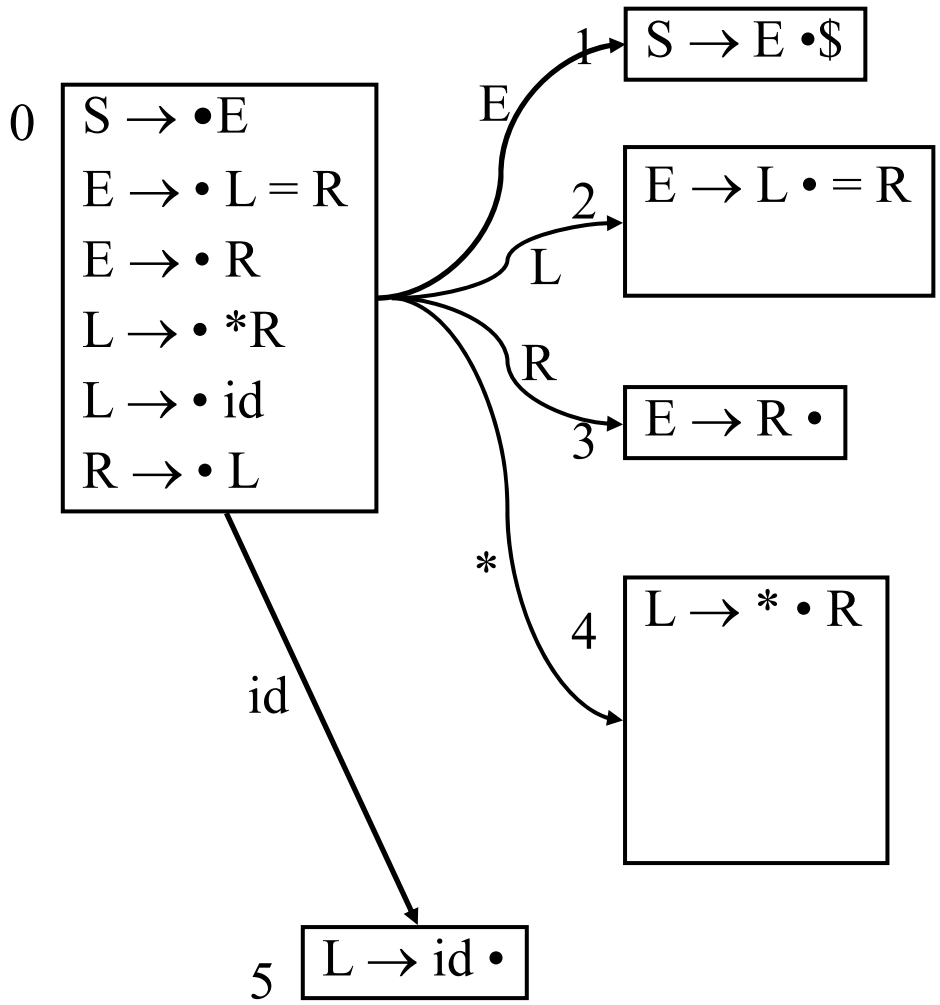
0 $S \rightarrow E\$$
1 $E \rightarrow L = R$
2 $E \rightarrow R$
3 $L \rightarrow id$
4 $L \rightarrow *R$
5 $R \rightarrow L$

0

$S \rightarrow \bullet E$
$E \rightarrow \bullet L = R$
$E \rightarrow \bullet R$
$L \rightarrow \bullet *R$
$L \rightarrow \bullet id$
$R \rightarrow \bullet L$

Example

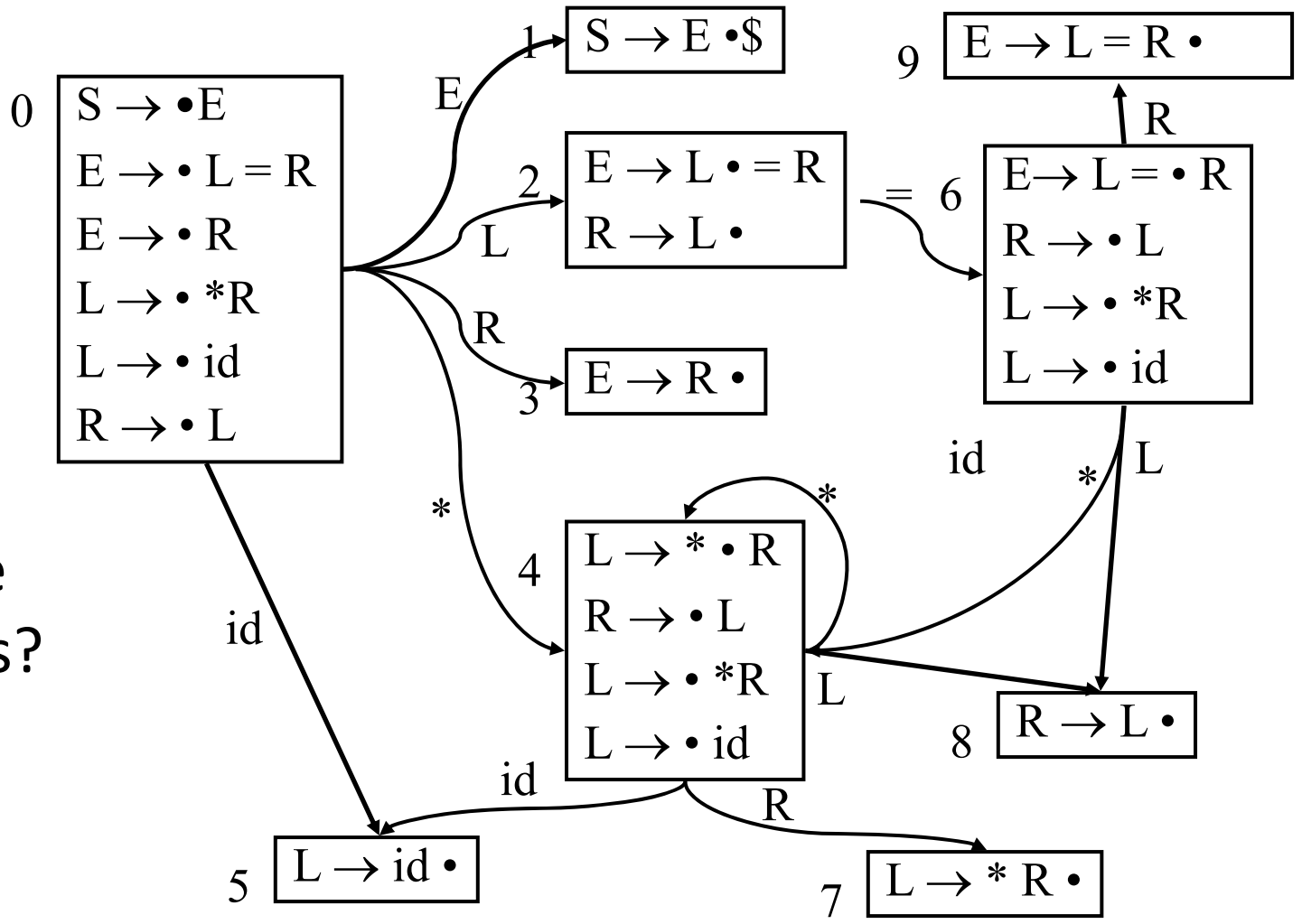
0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L



Example

0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

What are the reduce states?

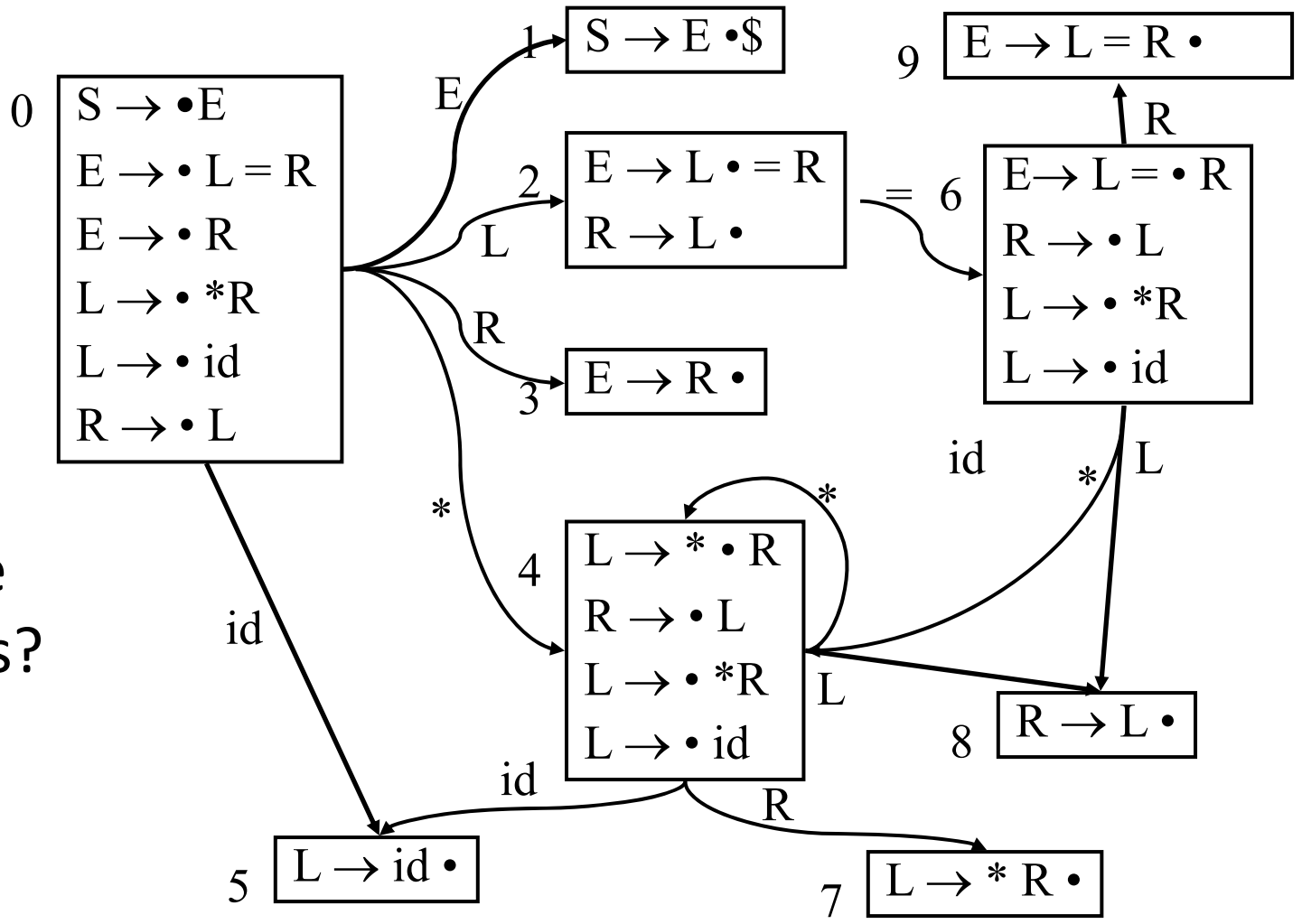


Example

0 $S \rightarrow E\$$
1 $E \rightarrow L = R$
2 $E \rightarrow R$
3 $L \rightarrow id$
4 $L \rightarrow *R$
5 $R \rightarrow L$

What are the reduce states?

1,2,3,5,7,8,9

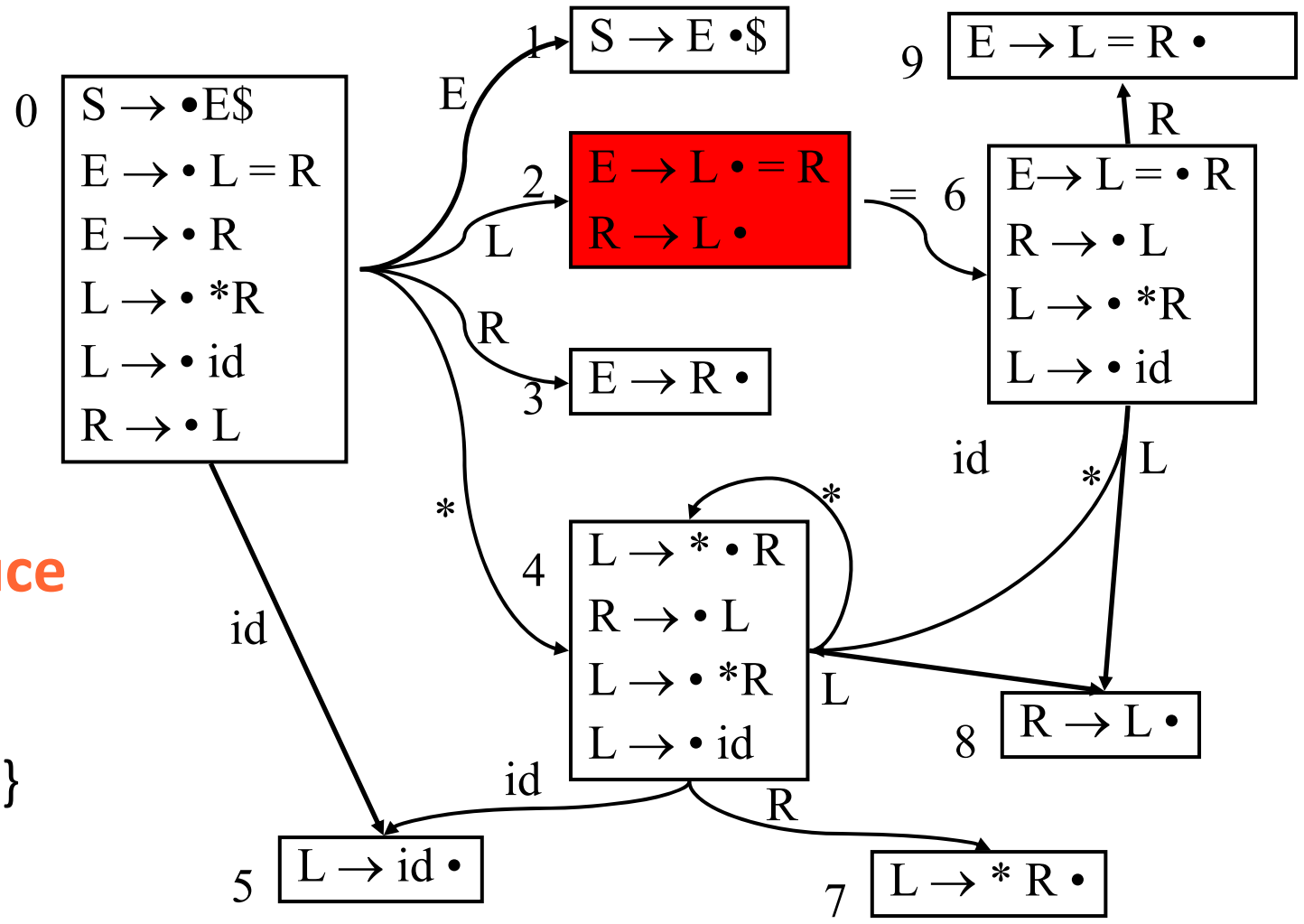


Example

0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

shift/reduce conflict

follow(R) = {=,$}



Problem with SLR

- Reduce on ALL terminals in FOLLOW set

S	→	L = R
	|	R
L	→	* R
	|	id
R	→	L

2 $S \rightarrow L \bullet = R$

$R \rightarrow L \bullet$

- FOLLOW(R) = FOLLOW(L)
- But, we should never reduce R → L on '='
  I.e., R=… is not a viable prefix for a right sentential form
- Thus, there should be no reduction in state 2
- How can we solve this?

LR(1) Items

- An LR(1) item is an LR(0) item combined with a single terminal (the *lookahead*)
- [X $\rightarrow \alpha \bullet \beta$, a] Means
  - α is at top of stack
  - Input string is derivable from βa
- In other words, when we reduce X $\rightarrow \alpha\beta$, a had better be the look ahead symbol.
- Or, Only put 'reduce by X $\rightarrow \alpha\beta$' in **`action[`**s,a**`]`**
- Can construct states as before, but have to modify closure

What LR(1) Items Mean

- [X $\rightarrow$ • α β γ, a]
  input is consistent with X $\rightarrow$ α β γ
- [X $\rightarrow$ α • β γ, a]
  input is consistent with X $\rightarrow$ α β γ and we have already recognized α
- [X $\rightarrow$ α β • γ, a]
  input is consistent with X $\rightarrow$ α β γ and we have already recognized α β
- [X $\rightarrow$ α β γ •, a]
  input is consistent with X $\rightarrow$ α β γ and if lookahead symbol is a, then we can reduce to X

LR(1) Closure

```
closure(state)
    repeat
       foreach item [A → α•Xβ, t] in state
          foreach production X → w
            and each terminal t′ in FIRST(βt)
             state.add([X → •w, t′])
    until state does not change
    return state
```

Closure

closure({S → •E$, ?}) =

S → •E$, ?

0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

FIRST($) = {$}

Closure

closure({S → •E$, ?}) =

S → •E$,	?
E → •L = R,	$
E → •R,	$

0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

FIRST($)	{$}
FIRST(=R$)	{=}

Closure

closure({S → •E$, ?}) =

S → •E$,	?
E → •L = R,	$
E → •R,	$
L → •*id*,	=
L → •*R,	=

0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

FIRST($)	{$}
FIRST(=R$)	{=}

Closure

closure({S → •E$, ?}) =

S → •E$,	?
E → •L = R,	$
E → •R,	$
L → •*id*,	=
L → •*R,	=
R → •L,	$

0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

FIRST($)	{$}
FIRST(=R$)	{=}

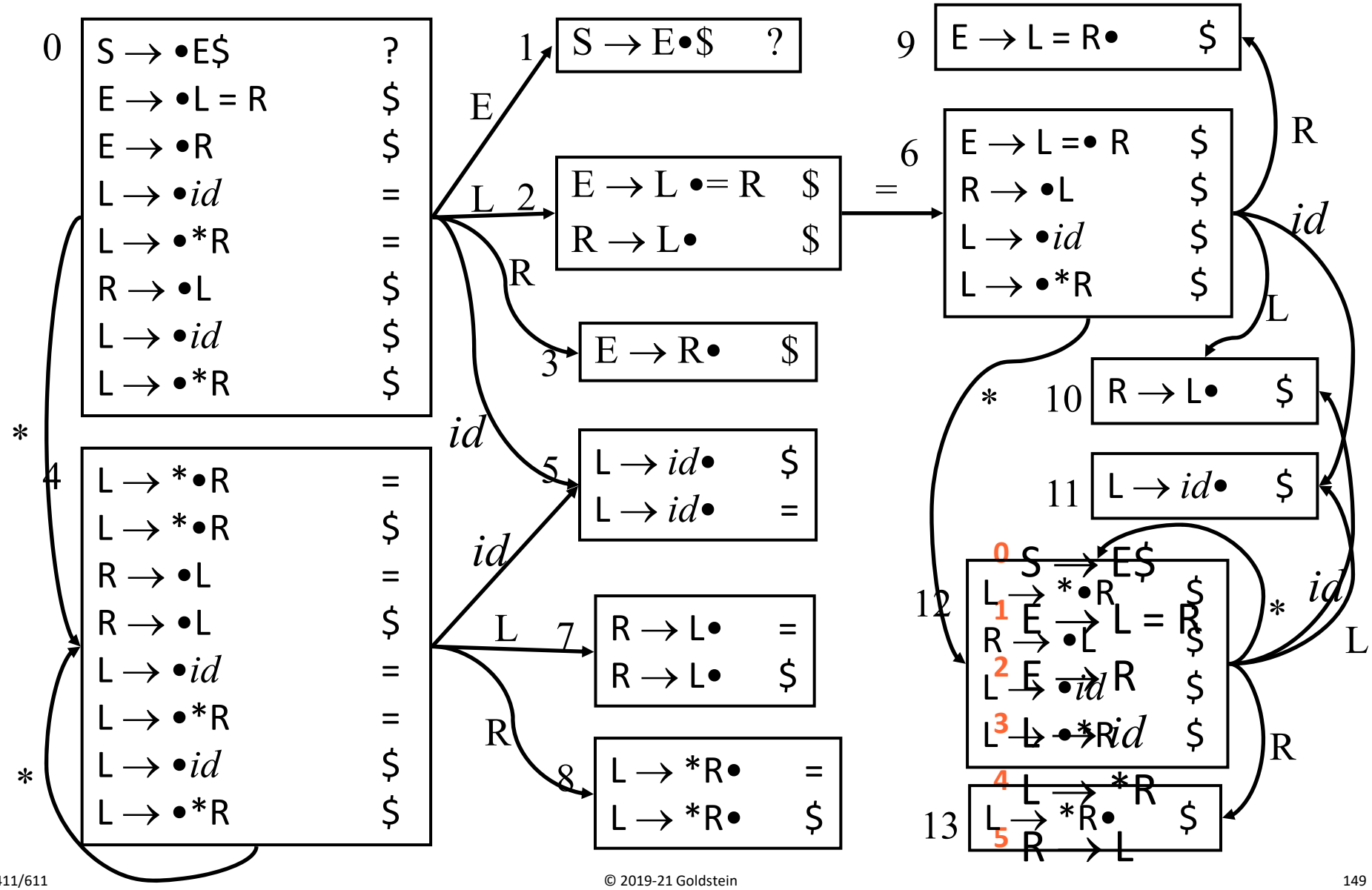
Closure

closure({S → •E$, ?}) =

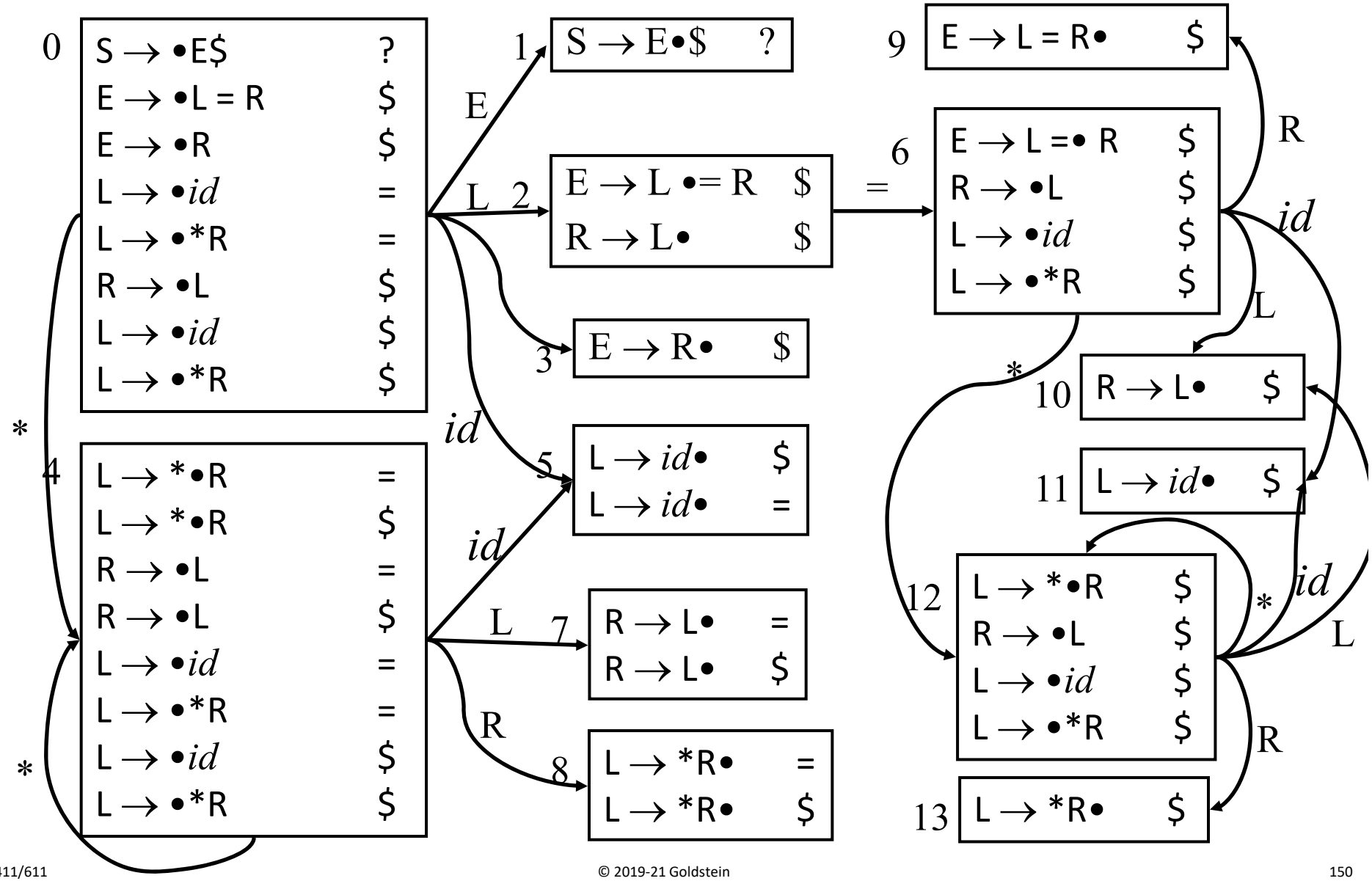
S → •E$,	?
E → •L = R,	$
E → •R,	$
L → •*id*,	=
L → •*R,	=
R → •L,	$
L → •*id*,	$
L → •*R,	$

0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

LR(1) Example



LR(1) Example



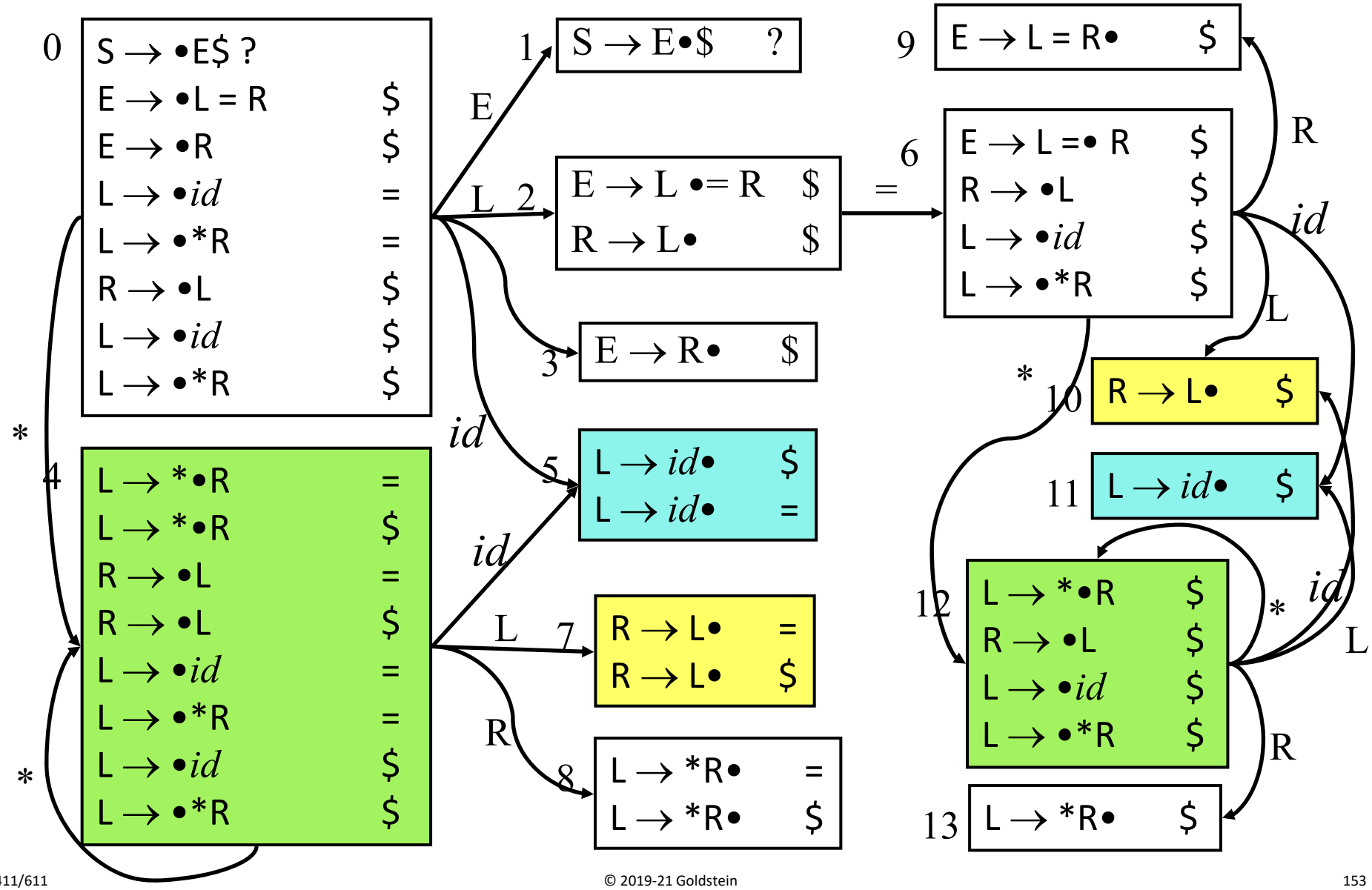
Parsing Table for LR(1)

- 14 states versus 10 LR(0) states
- In general, the number of states (and therefore size of the parsing table) is much larger with LR(1) items

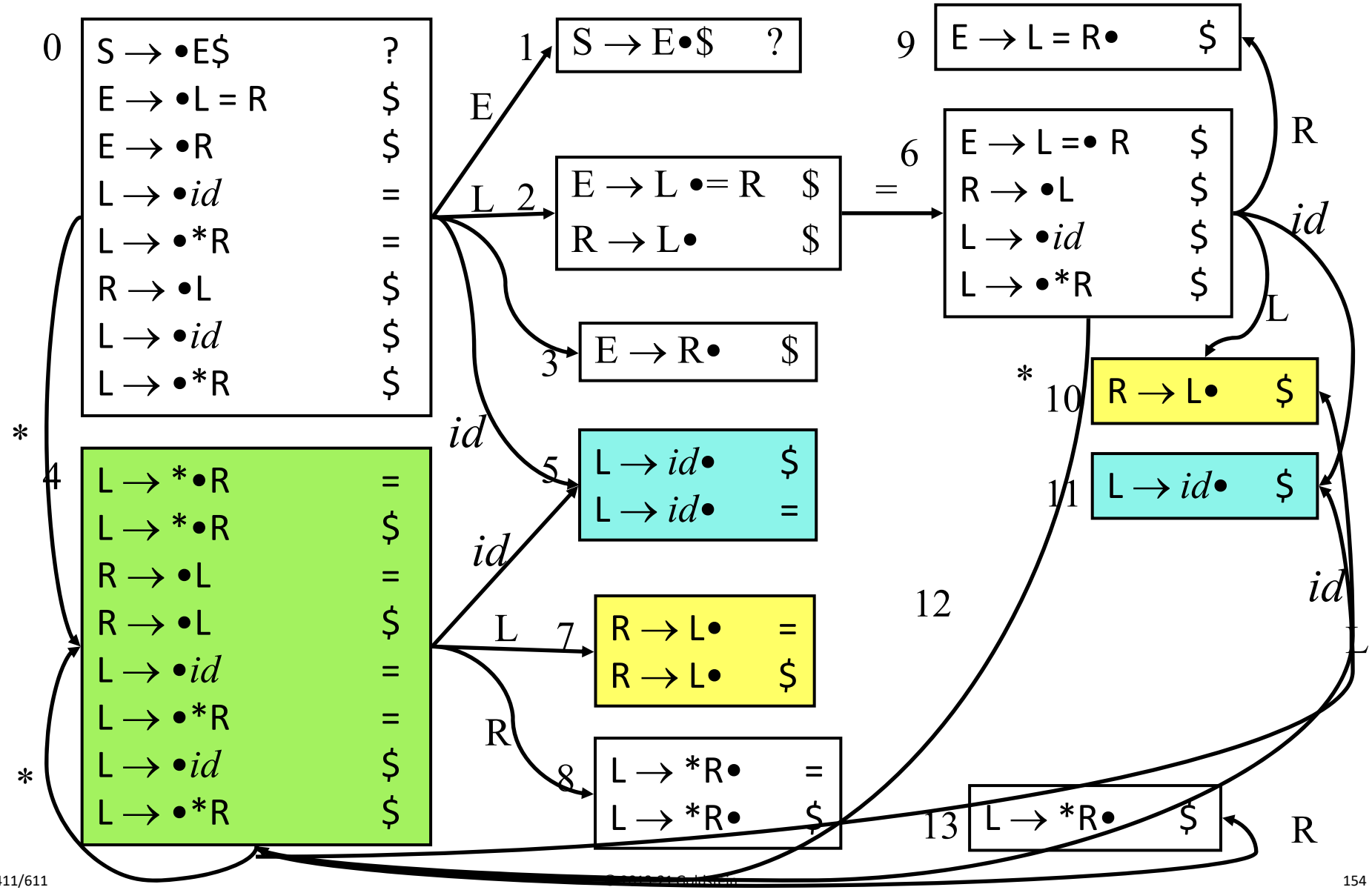
LALR: Lookahead LR

- More powerful than SLR
- Given LR(1) states, merge states that are identical except for lookaheads
- End up with same size table as SLR
- Can this introduce conflicts?

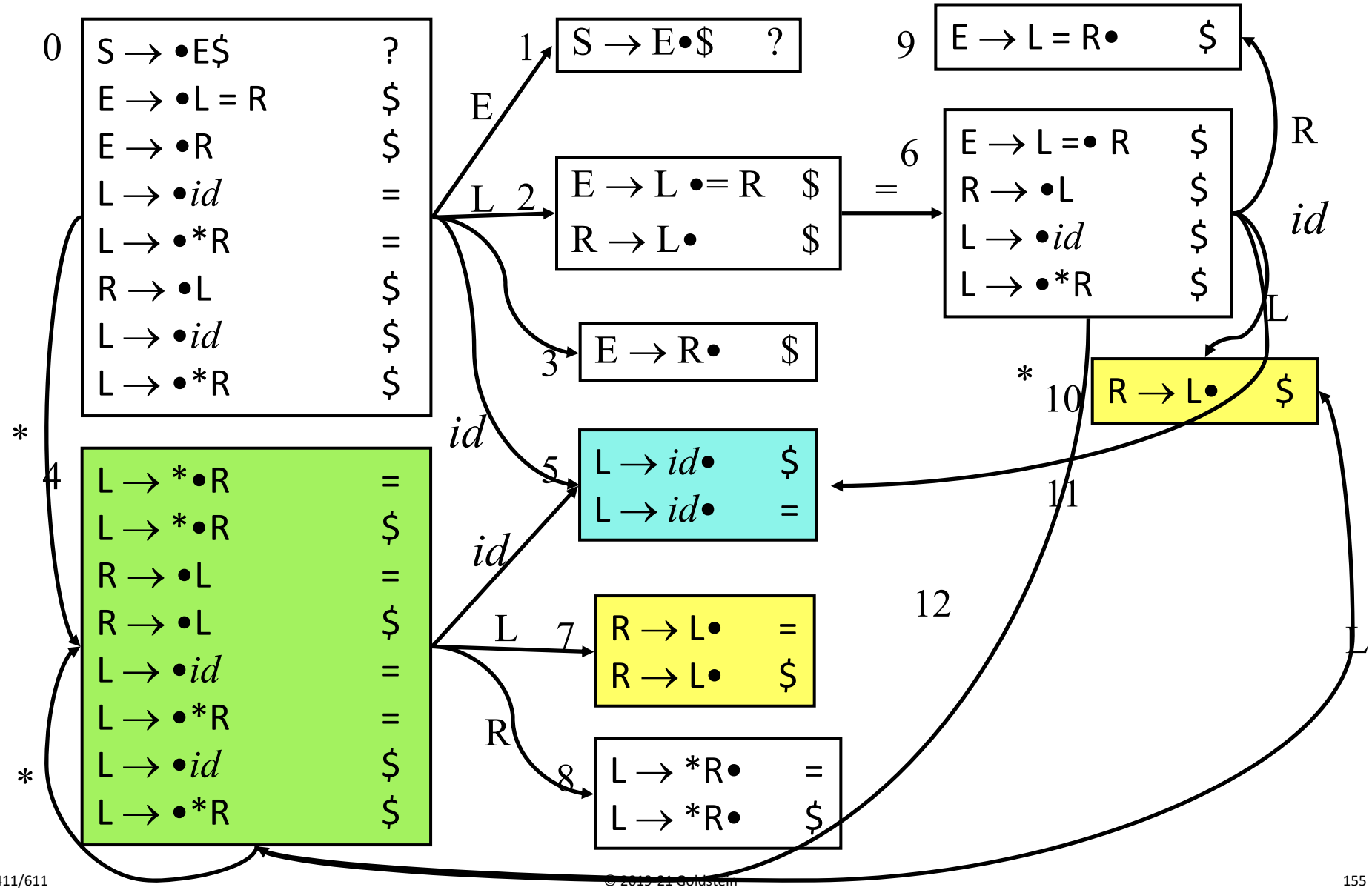
Merge-able states



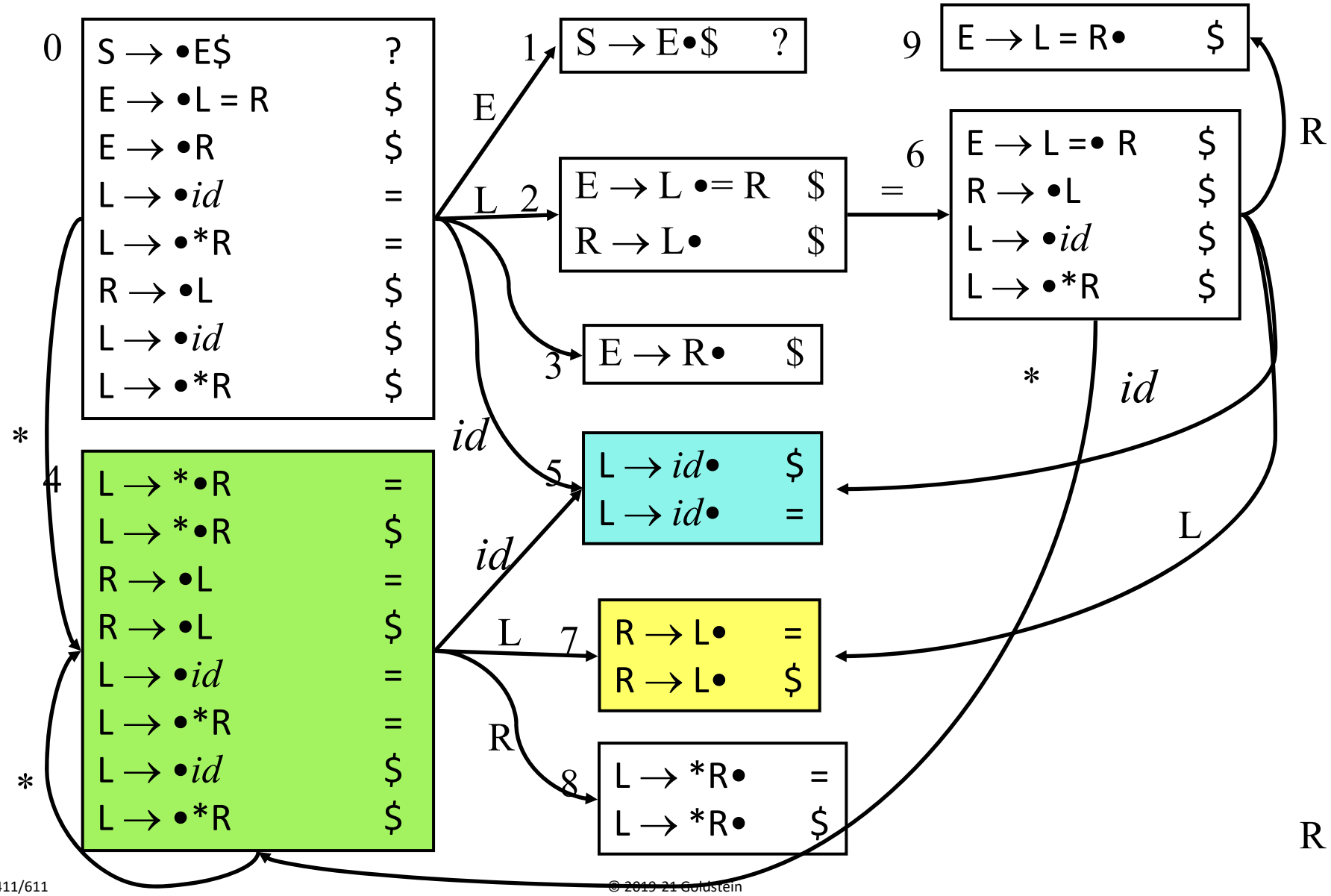
Merge-able states



Merge-able states



Merge-able states



LALR

- Can generate parse table without constructing LR(1) item sets
  - construct LR(0) item sets
  - compute *lookahead* sets
    - more precise than follow sets
- LALR is used by most parser generators (e.g., bison,lalrpop,ocamlyacc,…)

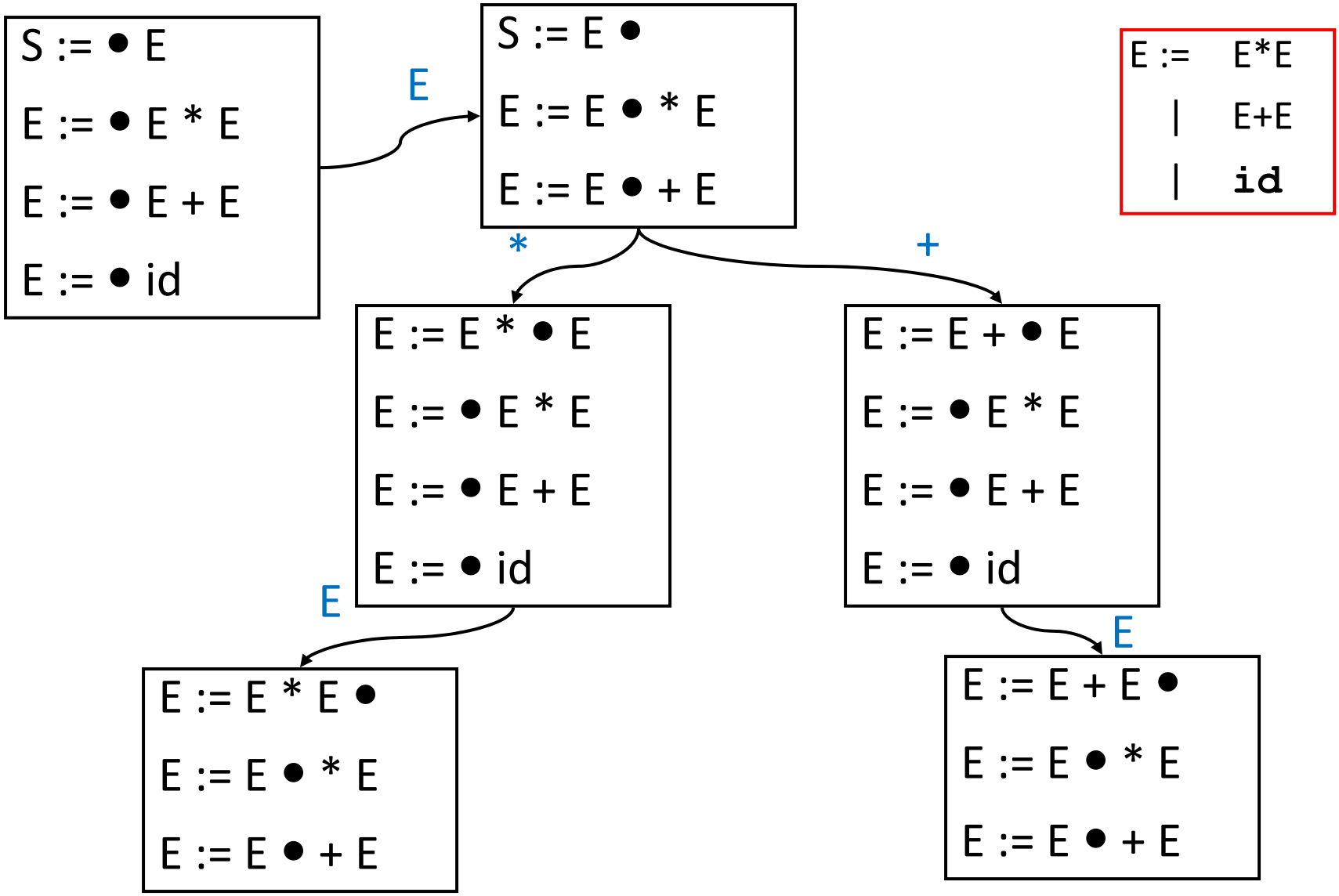
Recap

- LR(0) not very useful
- SLR uses follow sets to reduce
- LALR uses lookahead sets
- LR(1) uses full lookahead context

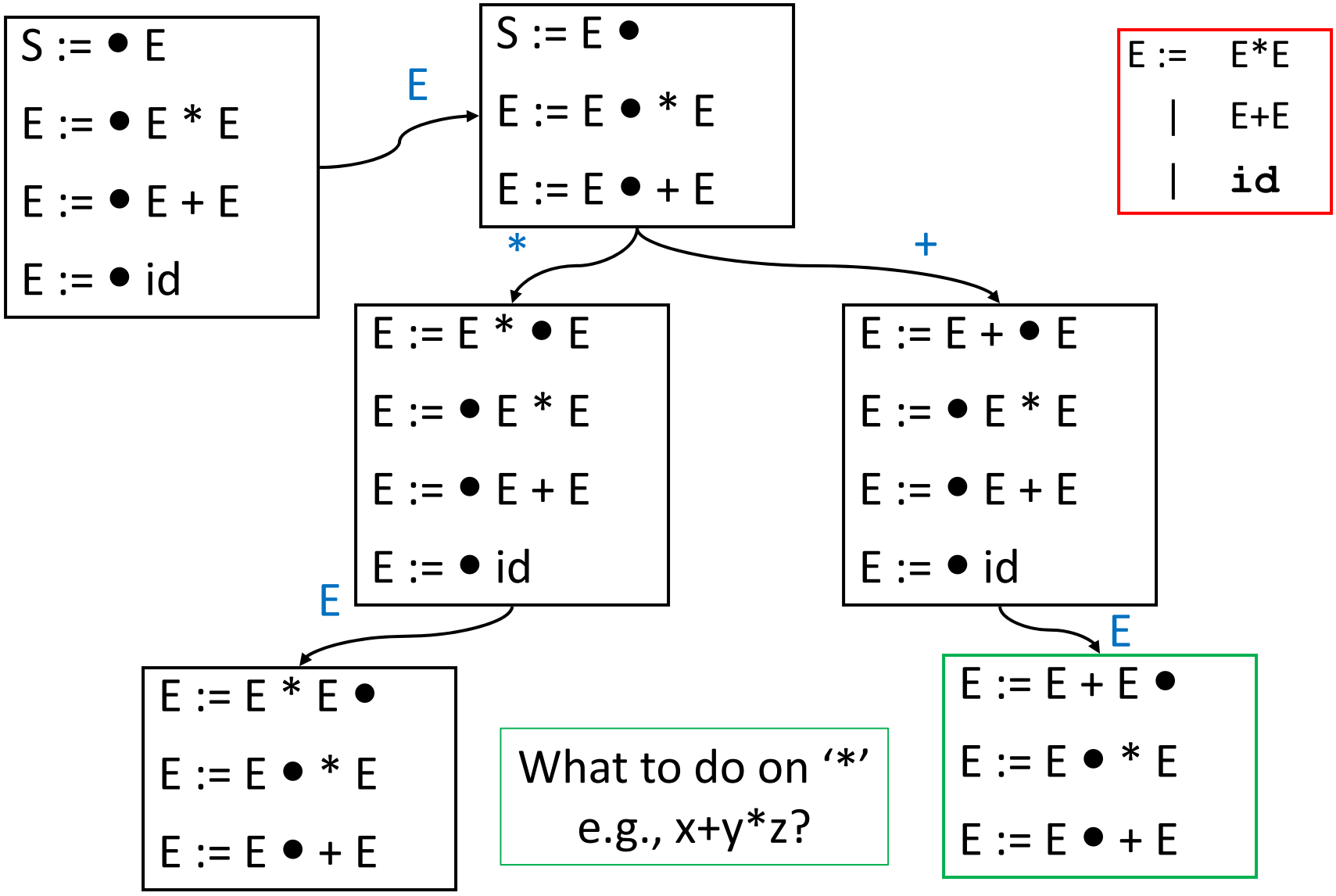
Power of shift-reduce parsers

- There are unambiguous grammars which cannot be parsed with shift-reduce parsers.
- Such grammars can have
  - shift/reduce conflicts
  - reduce/reduce conflicts
- There grammars are not LR(k)
- But, we can often choose shift or reduce to recognize what want.

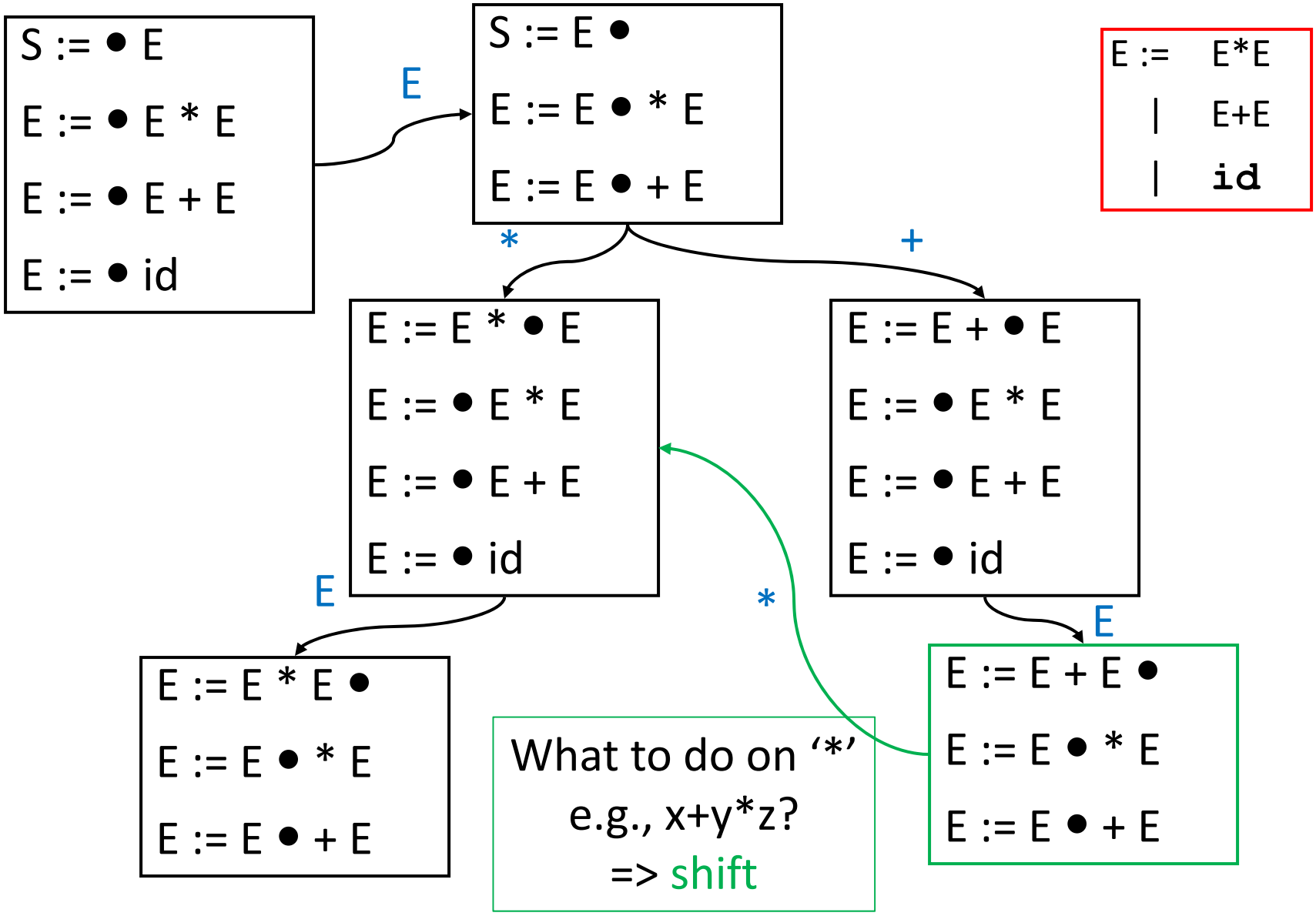
Expression Grammars & Precedence



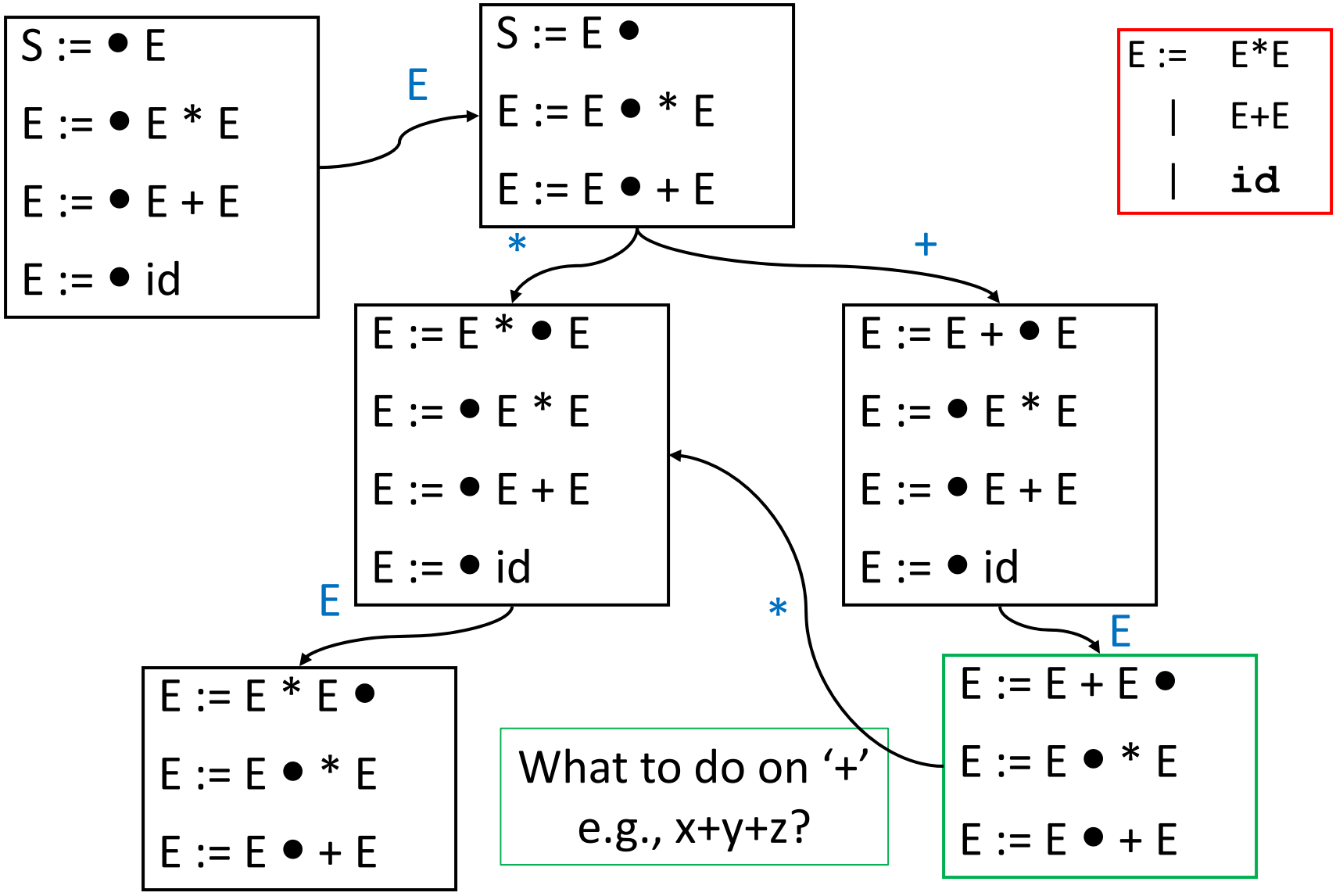
Expression Grammars & Precedence



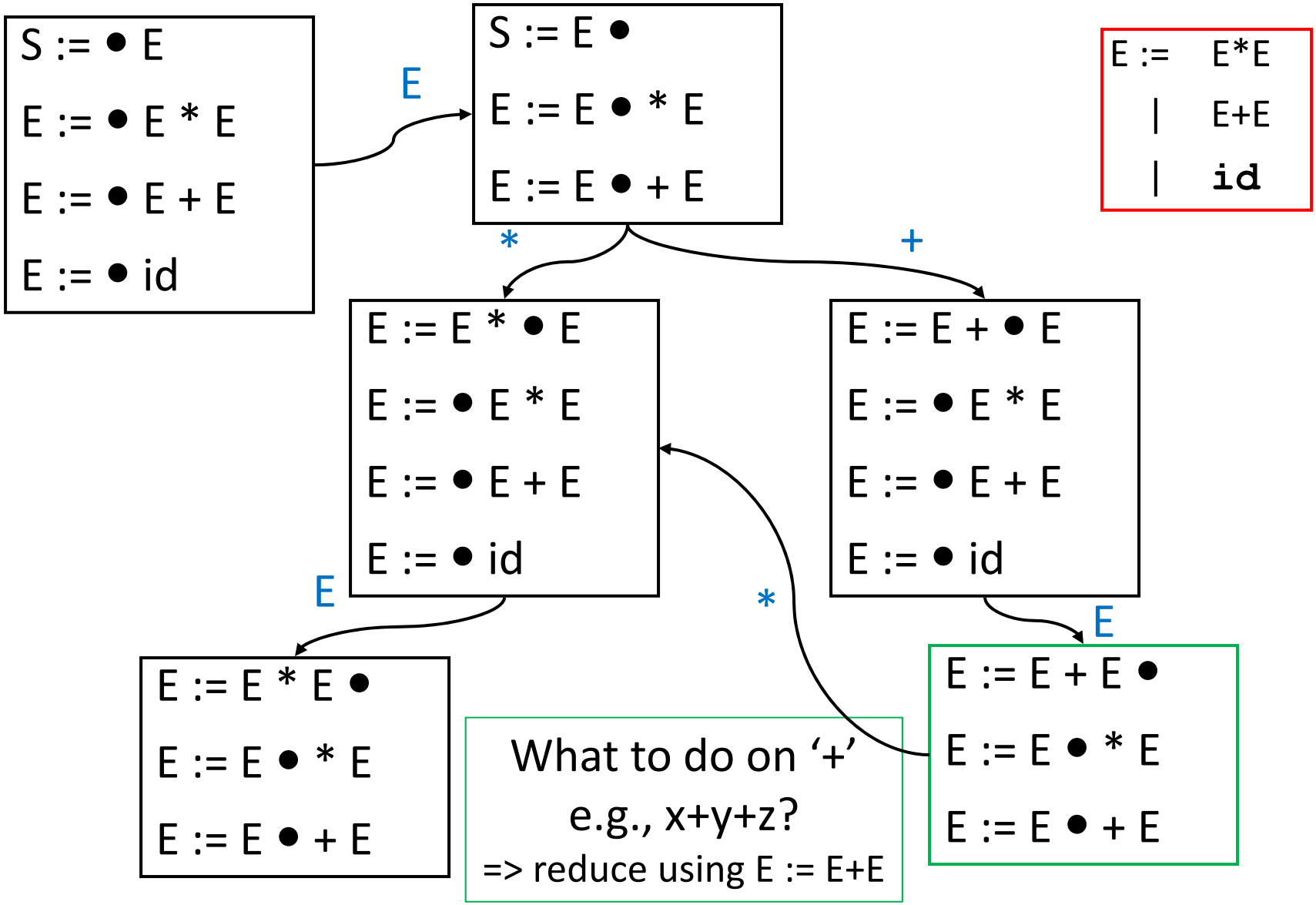
Expression Grammars & Precedence



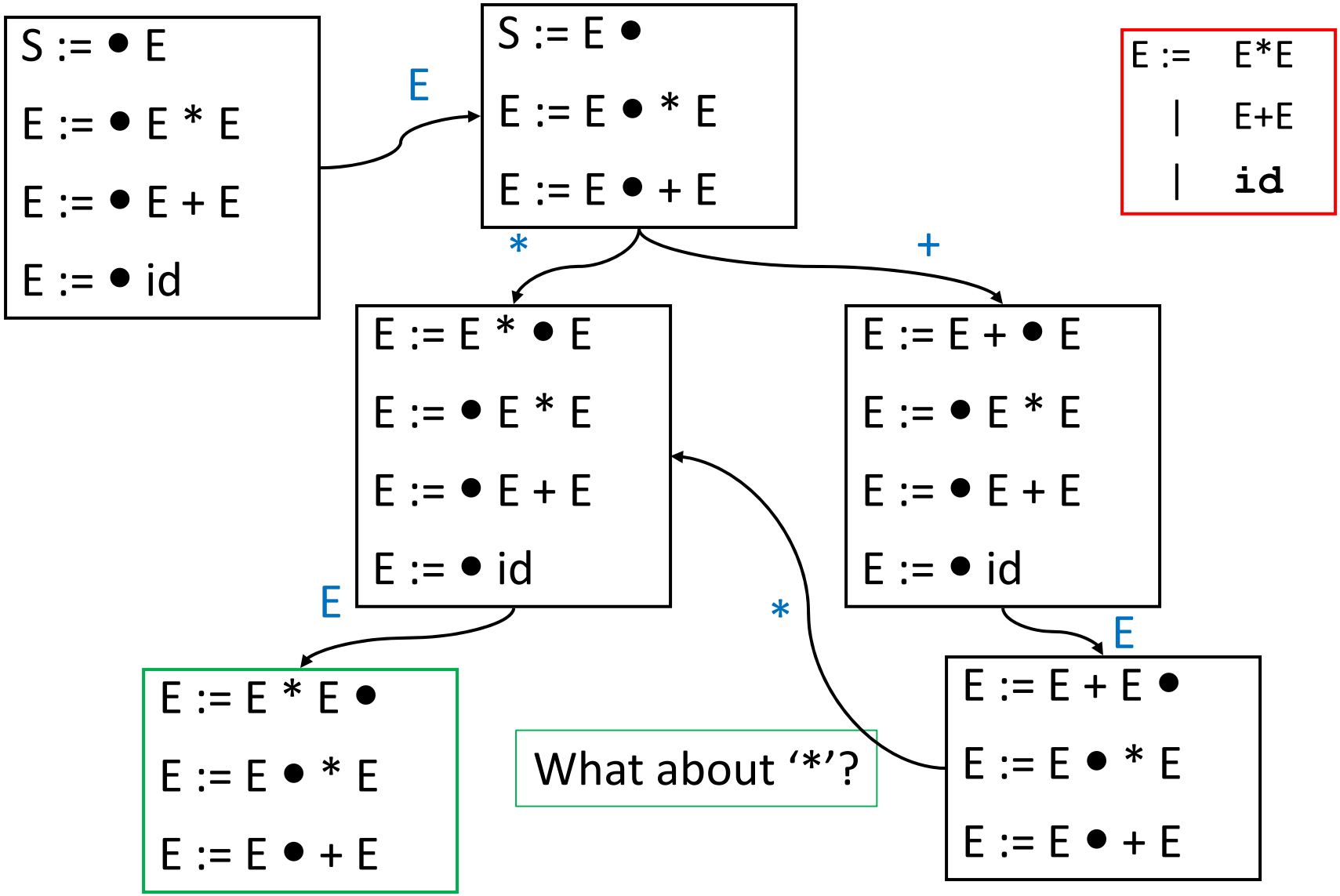
Expression Grammars & Precedence



Expression Grammars & Precedence



Expression Grammars & Precedence



Bison

- Precedence and Associativity declarations
- Precedence derived from order of directivies: from lowest to highest
- Associativity from %left, %right, %nonassoc
- Can be attached to rules as well (This can solve the dangling if-else problem
- Use output of generator showing items and transitions to debug s/r and r/r errors

Dangling Else

S := **if** E **then** S

| **if** E **then** S **else** S

| **other**

- We can be in the following state:

  … **if** E **then** S **else** … **$**

- What do we do?
  - shift the **else** (hoping to reduce by second rule), or
  - reduce by first rule