40-414 Compiler Design

Bottom-Up Parsing

Lecture 6

Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing

- Bottom-up is the preferred method

An Introductory Example

- Bottom-up parsers don't need left-factored grammars

- Revert to the "natural" grammar for our example:

  $E \rightarrow T + E \mid T$

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

- Consider the string: int * int + int

The Idea

Bottom-up parsing *reduces* a string to the start symbol by inverting productions:

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

Observation

- Read the productions in reverse (from bottom to top)
- This is a rightmost derivation!

RMD		PARSING	
↑	int * int + int	↓	T → int
	int * T + int		T → int * T
	T + int		T → int
	T + T		E → T
	T + E		E → T + E
	E		

Important Fact #1

Important Fact #1 about bottom-up parsing:

A bottom-up parser traces a rightmost derivation in reverse

A Bottom-up Parse

int * int + int

int * T + int

T + int

T + T

T + E

E

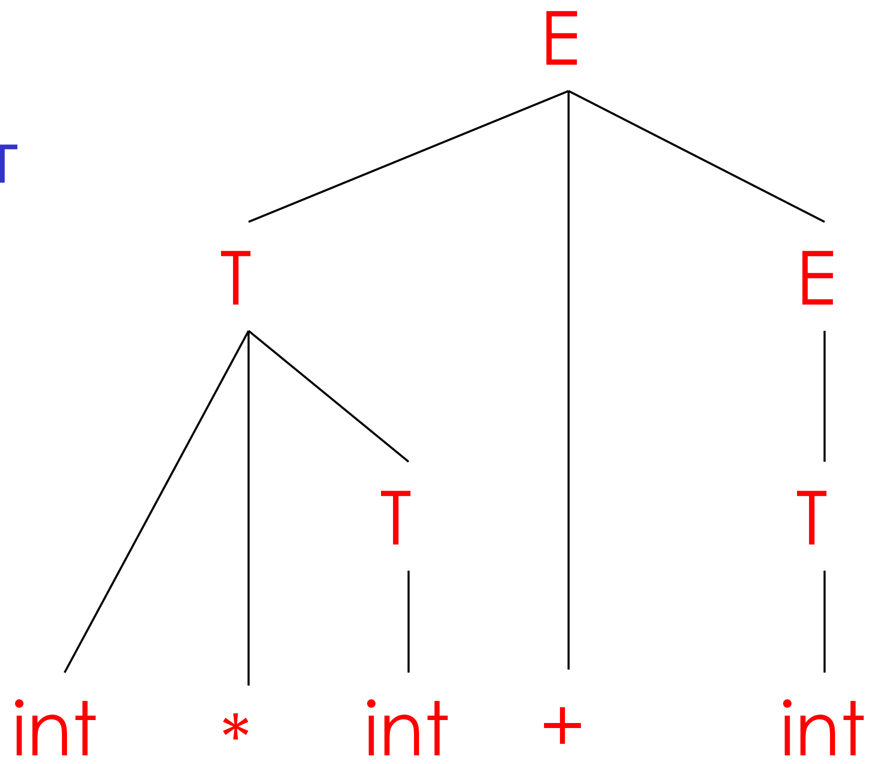
$E \rightarrow T + E$

$E \rightarrow T$

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

$T \rightarrow \text{int}$

$T \rightarrow (E)$



A Bottom-up Parse in Detail (1)

int * int + int

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow$ int * T

$T \rightarrow$ int

$T \rightarrow (E)$

int * int + int

A Bottom-up Parse in Detail (2)

int * int + int

int * T + int

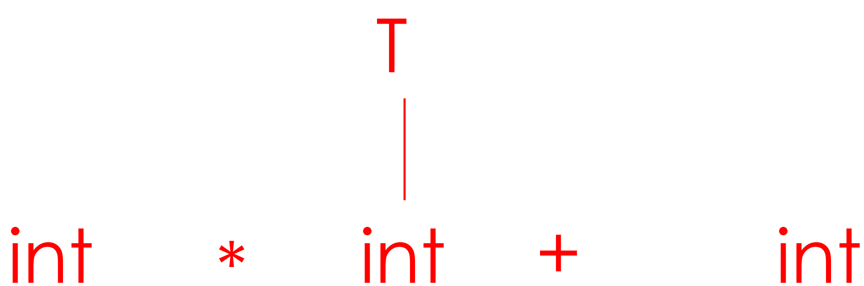
$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow$ int * T

$T \rightarrow$ int

$T \rightarrow (E)$



A Bottom-up Parse in Detail (3)

int * int + int

int * T + int

T + int

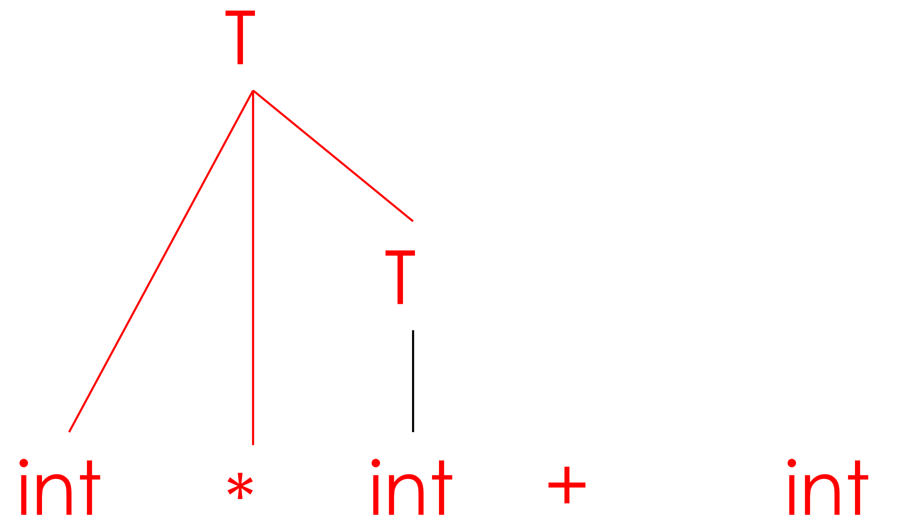
E → T + E

E → T

T → int * T

T → int

T → (E)



A Bottom-up Parse in Detail (4)

int * int + int

int * T + int

T + int

T + T

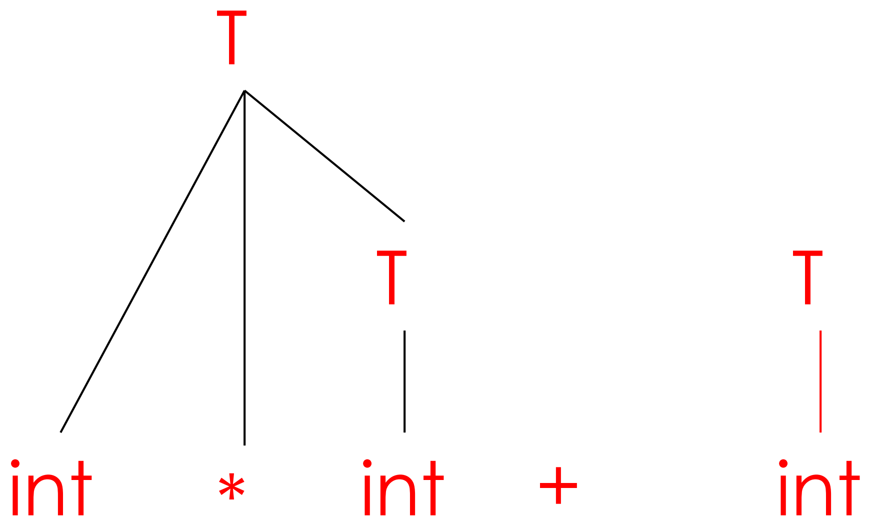
$E \rightarrow T + E$

$E \rightarrow T$

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

$T \rightarrow \text{int}$

$T \rightarrow (E)$



A Bottom-up Parse in Detail (5)

int * int + int

int * T + int

T + int

T + T

T + E

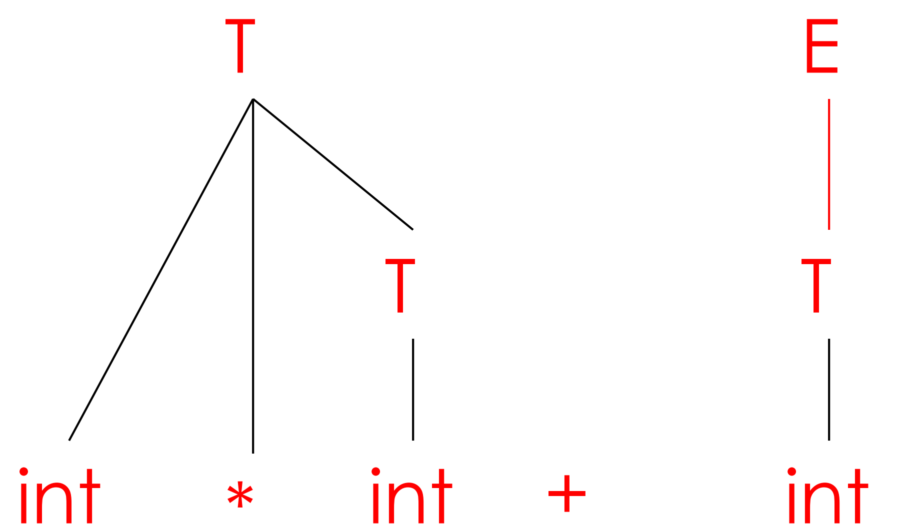
$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow int * T$

$T \rightarrow int$

$T \rightarrow (E)$



A Bottom-up Parse in Detail (6)

int * int + int

int * T + int

T + int

T + T

T + E

E

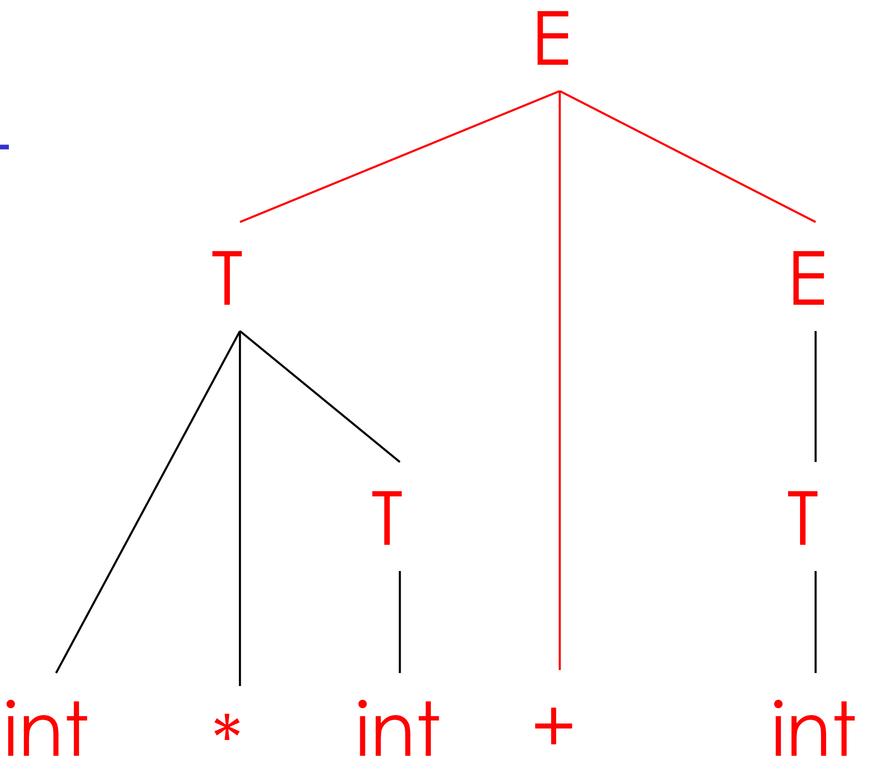
$E \rightarrow T + E$

$E \rightarrow T$

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

$T \rightarrow \text{int}$

$T \rightarrow (E)$



Where Do Reductions Happen?

Important Fact #1 has an interesting consequence:

- Let $\alpha\beta\omega$ be a step of a bottom-up parse
- Assume the next reduction is by $X \rightarrow \beta$
- Then ω is a string of terminals

Why? Because $\alpha X\omega \rightarrow \alpha\beta\omega$ is a step in a right-most derivation

Notation

- Idea: Split string into two substrings
  - Right substring is as yet unexamined by parsing (a string of terminals)
  - Left substring has terminals and non-terminals

- The dividing point is marked by a | $\alpha\beta \,|\, \omega$
  - The | is not part of the string

- Initially, all input is unexamined $|x_1x_2 \ldots x_n$

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

Shift

- *Shift*: Move | one place to the right
  - Shifts a terminal to the left string

$$ABC|\underbrace{xyz}_{} \Rightarrow ABCx|yz$$

Reduce

- Apply an inverse production at the right end of the left string
  - If $A \rightarrow xy$ is a production, then

$$Cbxy|ijk \Rightarrow CbA|ijk$$

The Example with Reductions Only

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

The Example with Shift-Reduce Parsing

|int * int + int	shift
int | * int + int	shift
int * | int + int	shift
int * int | + int	reduce T → int
int * T | + int	reduce T → int * T
T | + int	shift
T + | int	shift
T + int |	reduce T → int
T + T |	reduce E → T
T + E |	reduce E → T + E
E |	

A Shift-Reduce Parse in Detail (1)

|int * int + int

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow int * T$

$T \rightarrow int$

$T \rightarrow (E)$

int * int + int

↑

A Shift-Reduce Parse in Detail (2)

|int * int + int

int | * int + int

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow$ int * T

$T \rightarrow$ int

$T \rightarrow (E)$

int * int + int

↑

A Shift-Reduce Parse in Detail (3)

| int * int + int
int | * int + int
int * | int + int

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow int * T$

$T \rightarrow int$

$T \rightarrow (E)$

int * int + int

↑

A Shift-Reduce Parse in Detail (4)

|int * int + int

int | * int + int

int * | int + int

int * int | + int

$E \to T + E$

$E \to T$

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

$T \to \text{int}$

$T \to (E)$

int * int + int

↑

A Shift-Reduce Parse in Detail (5)

| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

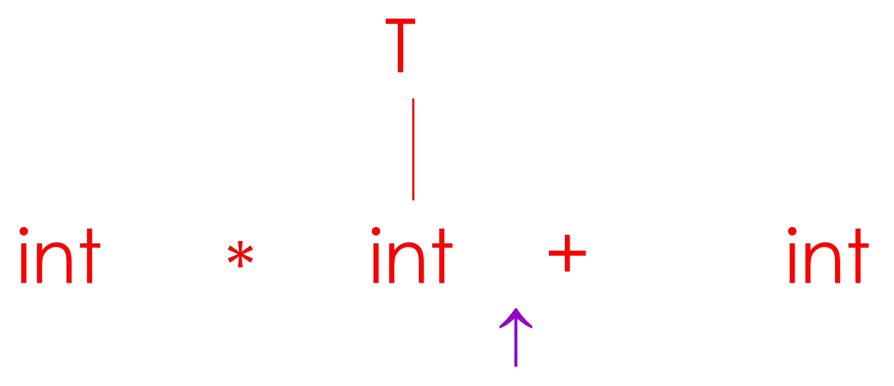
$E \rightarrow T + E$

$E \rightarrow T$

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

$T \rightarrow \text{int}$

$T \rightarrow (E)$



A Shift-Reduce Parse in Detail (6)

|int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int

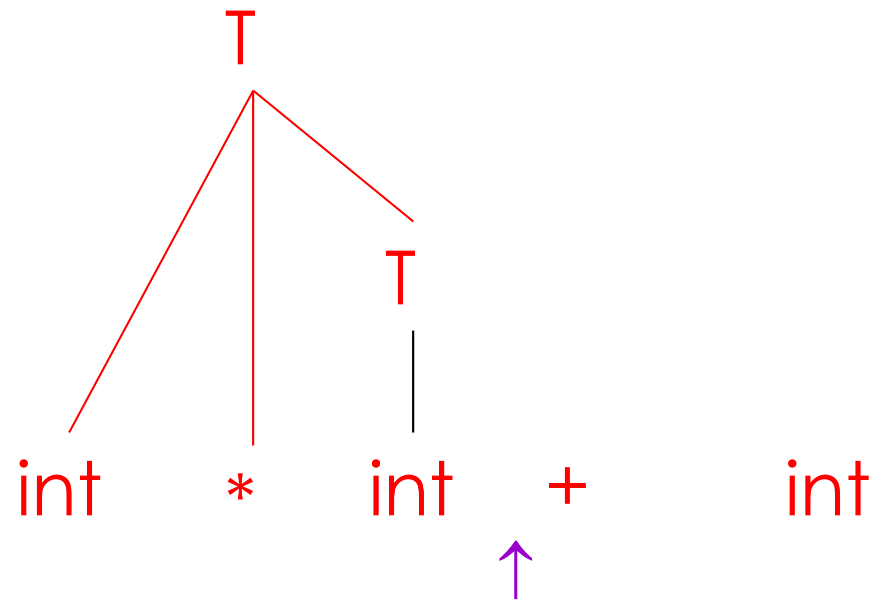
$E \rightarrow T + E$

$E \rightarrow T$

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

$T \rightarrow \text{int}$

$T \rightarrow (E)$



A Shift-Reduce Parse in Detail (7)

| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int

T + | int

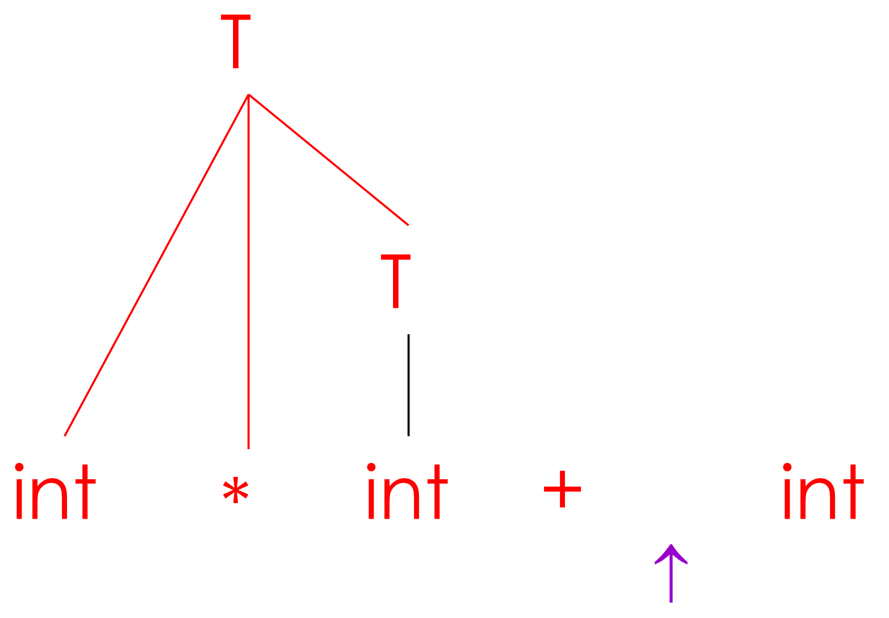
$E \rightarrow T + E$

$E \rightarrow T$

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

$T \rightarrow \text{int}$

$T \rightarrow (E)$



A Shift-Reduce Parse in Detail (8)

|int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int

T + | int

T + int |

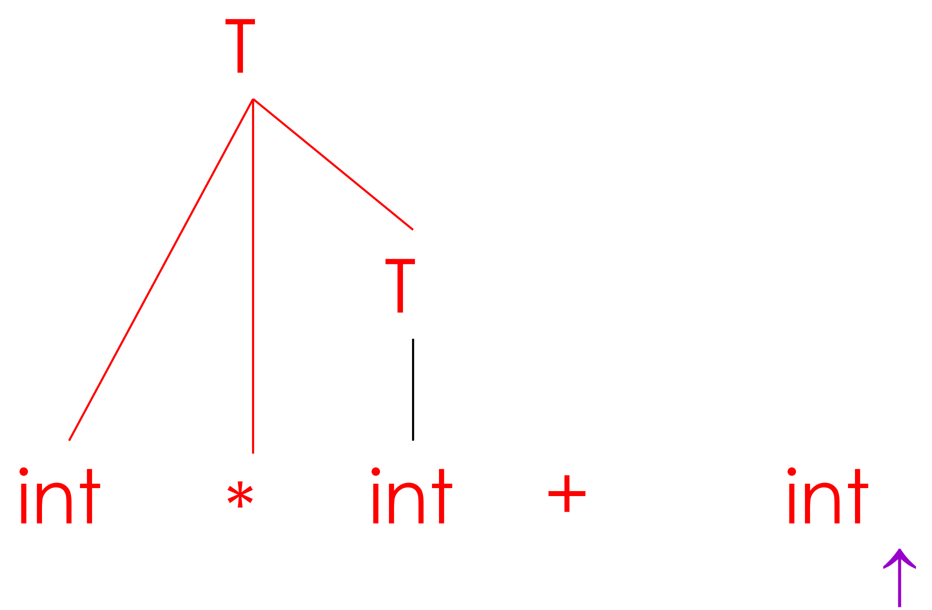
$E \to T + E$

$E \to T$

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

$T \to \text{int}$

$T \to (E)$



A Shift-Reduce Parse in Detail (9)

| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int

T + | int

T + int |

T + T |

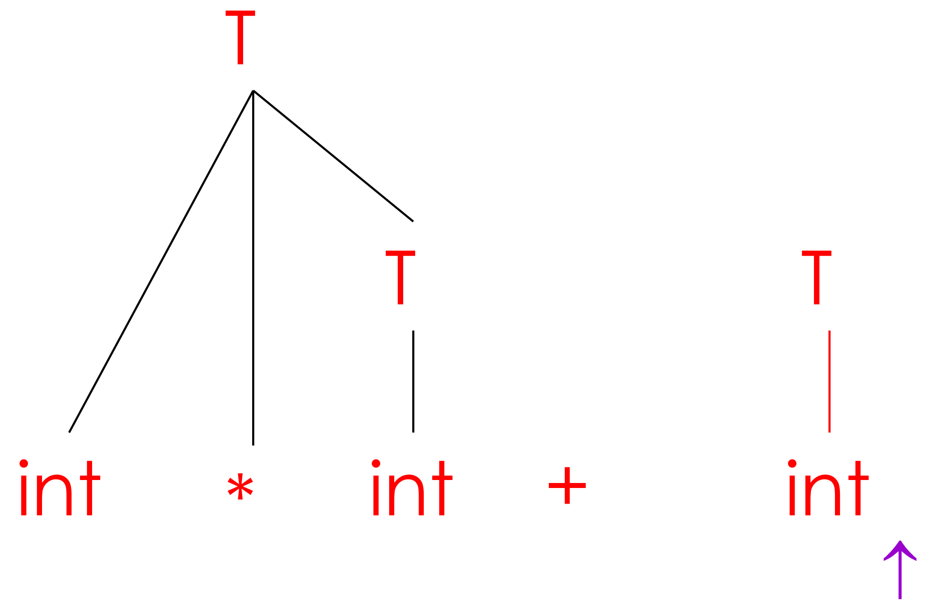
$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow int * T$

$T \rightarrow int$

$T \rightarrow (E)$



A Shift-Reduce Parse in Detail (10)

| 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 |

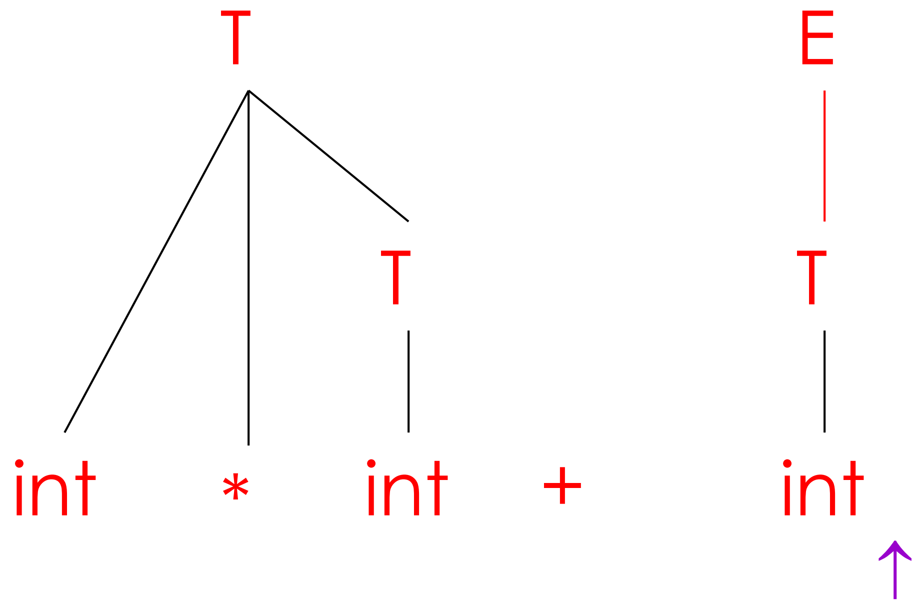
E → T + E

E → T

T → int * T

T → int

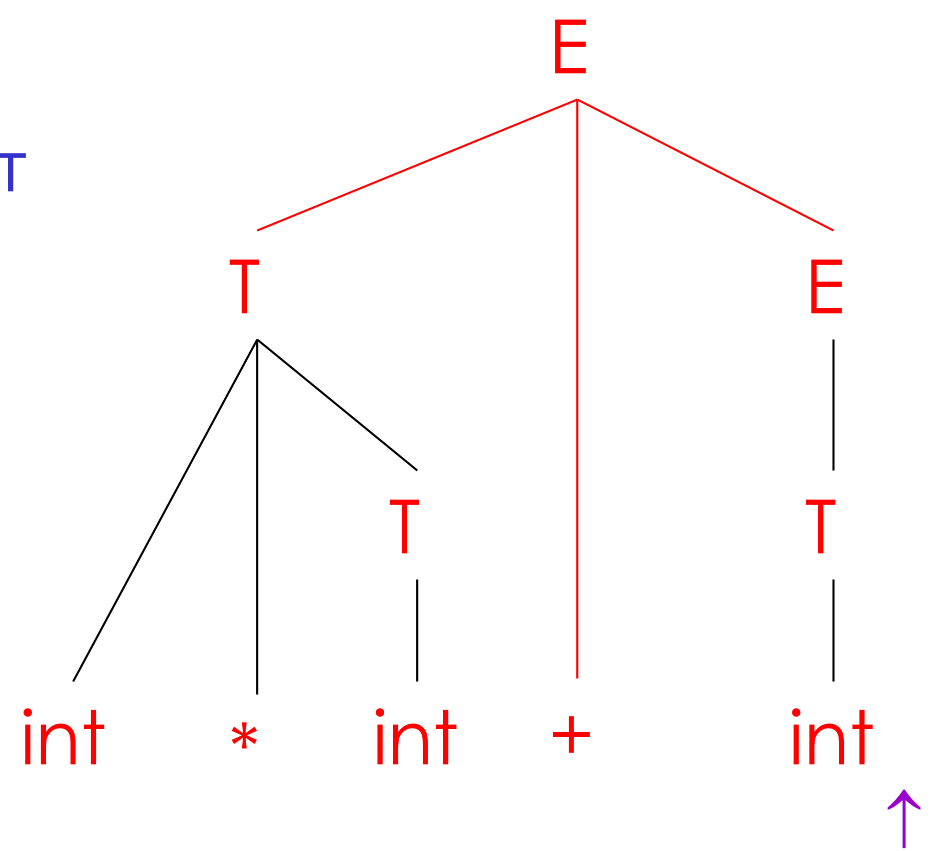
T → (E)



A Shift-Reduce Parse in Detail (11)

| 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 |

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



The Stack

- Left string can be implemented by a stack
  - Top of the stack is the |

- Shift pushes a terminal on the stack

- Reduce pops 0 or more symbols off of the stack (production rhs) and pushes a non-terminal on the stack (production lhs)

Conflicts

- In a given state, more than one action (shift or reduce) may lead to a valid parse

- If it is legal to shift or reduce, there is a *shift-reduce* conflict

- If it is legal to reduce by two different productions, there is a *reduce-reduce* conflict

Key Issue

- How do we decide when to shift or reduce?

- Example grammar:

  $E \to T + E \mid T$

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

- Consider step int | * int + int
  - We could reduce by $T \to \text{int}$ giving T | * int + int
  - A fatal mistake!
    - No way to reduce to the start symbol E
    - No production starts with T *

Handles

- Intuition: Want to reduce only if the result can still be reduced to the start symbol

- Assume a rightmost derivation

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

reduction

- Then $X \rightarrow \beta$ in the position after α is a *handle* of $\alpha\beta\omega$

Handles (Cont.)

- Formally:

- A *phrase* is a substring of a sentential form derived from exactly one non-terminal

- A *simple phrase* is a phrase created in one step

- A *handle* is a simple phrase of a right sentential form
  - i.e., $X \rightarrow \beta$ is a handle of $\alpha\beta\omega$, where ω is a string of terminals, if:

$$S \rightarrow \alpha X\omega \rightarrow \alpha\beta\omega$$

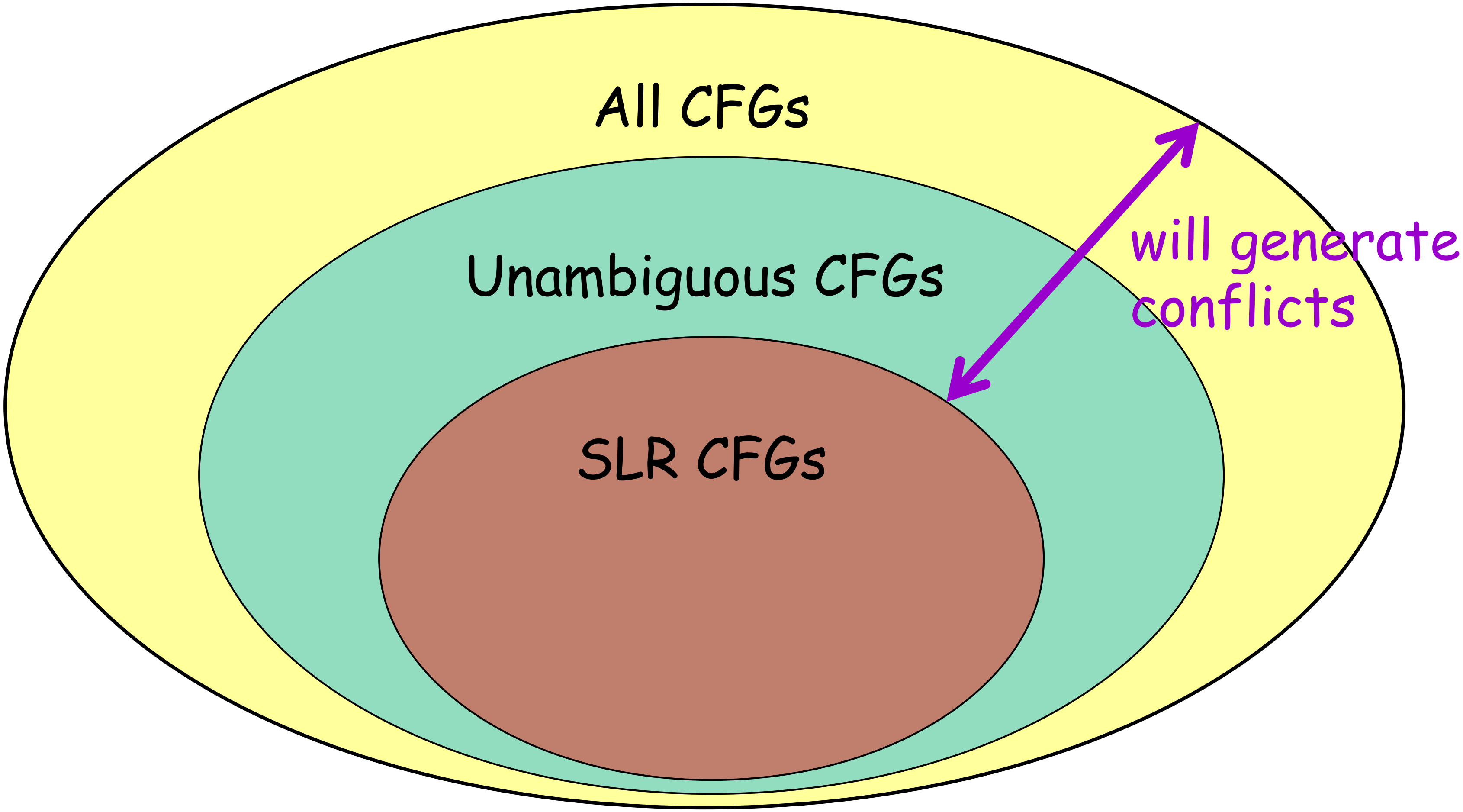
Handles (Cont.)

- Handles formalize the intuition
  - A handle is a string that can be reduced and also allows further reductions back to the start symbol (using a particular production at a specific spot)

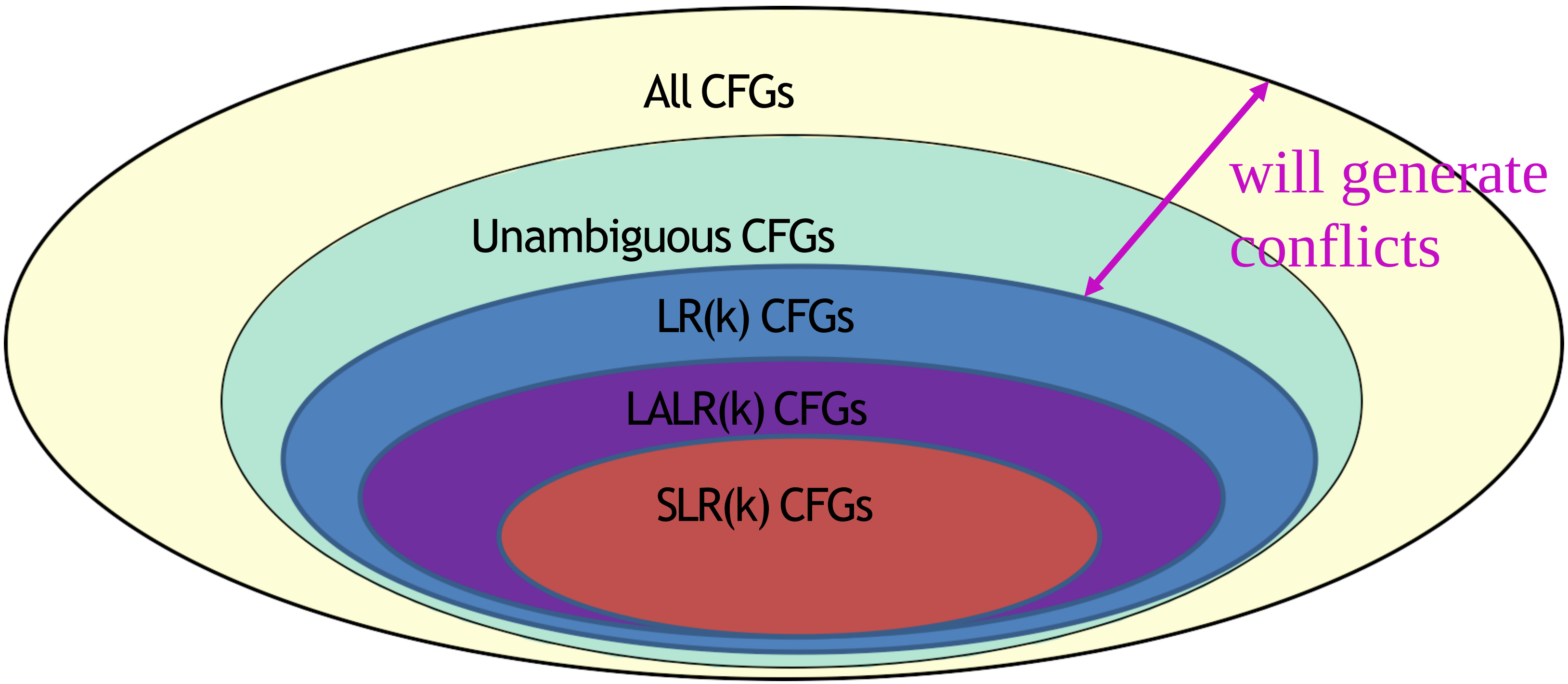
- We only want to reduce at handles

- Note: We have said what a handle is, not how to find handles

Grammars


All CFGs
Unambiguous CFGs
SLR CFGs
will generate
conflicts

Grammars


All CFGs
Unambiguous CFGs
LR(k) CFGs
LALR(k) CFGs
SLR(k) CFGs
will generate
conflicts

Bottom-Up Parsing

LR(k) parsing.

left to right scanning

right-most derivation

k lookahead
(k is omitted ➔ it is 1)

- LR parsing is the most general predictive parsing, yet it is still very efficient.
- The class of LR grammars is a proper superset of LL(1) grammars.

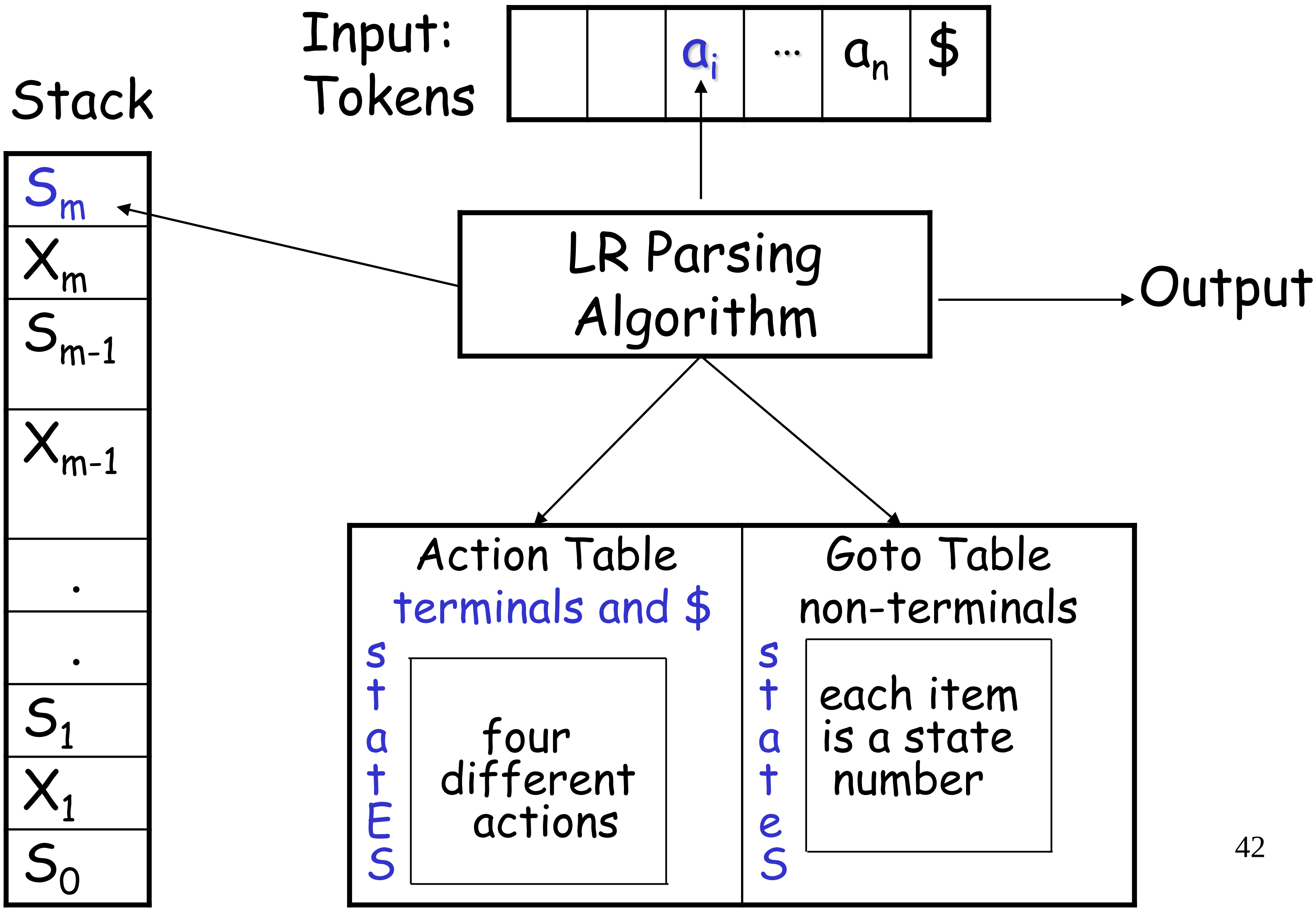
$$\text{LL(1)-Grammars} \subset \text{LR(1)-Grammars}$$

- An LR-parser can detect a syntactic error as soon as it is possible.

LR (k) parsing.

- SLR – Simple LR parser
- LR – most general LR parser
- LALR – intermediate LR parser (Look-Ahead LR)
- SLR, LR, and LALR work exactly the same; only their parse tables are different.

LR Parsing Algorithm



Configuration of LR Parsing

- A configuration in LR parsing is:

$$\underbrace{S_o\ X_1\ S_1 \ldots X_m\ S_m}_{\text{Stack}} \mid \underbrace{a_i\ a_{i+1} \ldots a_n\ \$}_{\text{Rest of Input}}$$

- S_m and a_i decides the parser action by consulting the parsing (action) table.

- *Initial Stack* contains just S_o

- A configuration of a LR parsing represents the right sentential form:

$$X_1 \ldots X_m\ a_i\ a_{i+1} \ldots a_n\ \$$$

Actions of A LR-Parser

1. shift s -- shifts the next token and the state s onto the stack

   $S_o\ X_1\ S_1 \ldots X_m\ S_m \mid a_i\ a_{i+1} \ldots a_n\ \$ \Rightarrow S_o\ X_1\ S_1 \ldots X_m\ S_m\ a_i\ s \mid a_{i+1} \ldots a_n\ \$$

2. reduce $A \rightarrow \beta$ (or reduce n, where n is a production number)
   - pop $2^*|\beta|$ (=r) items from the stack,
   - then push A and s, where s = goto[s_{m-r}, A]

   $S_o\ X_1\ S_1 \ldots X_m\ S_m \mid a_i\ a_{i+1} \ldots a_n\ \$ \Rightarrow S_o\ X_1\ S_1 \ldots X_{m-r}\ S_{m-r}\ A\ s \mid a_i \ldots a_n\ \$$

   - Parser output is the reducing production: $A \rightarrow \beta$

3. Accept – Parsing successfully completed

4. Error -- Parser detected an error (an empty entry in the action table)

SLR(1) Parsing Table Example

1 $E \to T$

2 $E \to T + E$

3 $T \to (E)$

4 $T \to int * T$

5 $T \to int$

Follow (E) = {\$,)}

Follow (T) = {\$,), +}

	Action							Goto	
state	int	+	*	(	)	$		E	T
0	S4			S3				1	2
1						acc			
2		S5			R1	R1			
3	S4			S3				7	2
4		R5	S6		R5	R5			
5	S4			S3				8	2
6	S4			S3					9
7					S10				
8					R2	R2			
9		R4			R4	R4			
10		R3			R3	R3			

LR(1) Parsing Example

stack	input	action
0	int*int$	Shift 4
0 int 4	*int$	Shift 6
0 int 4 * 6	int$	Shift 4
0 int 4 * 6 int 4	$	Reduce 5
0 int 4 * 6 T 9	$	Reduce 4
0 T 2	$	Reduce 1
0 E 1	$	Accept

Handle

1 E → T

2 E → T + E

3 T → (E)

4 T → int * T

5 T → int

state	int	+	*	(	)	$	E	T
0	S4			S3			1	2
1						acc		
2		S6			R1	R1		
4		R5	S6		R5	R5		
6	S4			S3				9
9		R4			R4	R4		

Some of the rows!

Constructing SLR(1) Parsing Table

- LR(0) items
- Augmented Grammar
- Closure operation
- GOTO function
- SLR(1) Transition Diagram

Items

- An item is a production with a "•" somewhere on the rhs

- The items for $T \rightarrow (E)$ are

  $T \rightarrow \bullet\, (E)$

  $T \rightarrow (\bullet E)$

  $T \rightarrow (E \bullet)$

  $T \rightarrow (E) \bullet$

Items (Cont.)

- The only item for $X \to \varepsilon$ is $X \to \bullet$

- Items are often called "LR(0) items"

Augmented Grammar

- To construct the SLR(1) Parsing Table, we start by adding a new rule $S' \rightarrow S$ to the grammar, where S' is a new non-terminal and S is the start symbol

- The new grammar is called an "Augmented Grammar"

The Closure Operation

- If I is a set of LR(0) items for a grammar G, then closure(I) is the set of LR(0) items constructed from I by the two rules:

1. Initially, every LR(0) item in I is added to closure(I).

2. If $A \rightarrow \alpha \bullet B \beta$ is in closure(I) and $B \rightarrow \gamma$ is a production rule of G;
then $B \rightarrow \bullet \gamma$ will be in closure(I). We will apply this rule until no more new LR(0) items can be added to closure(I).

The Closure Operation - Example

Grammar:

$S' \rightarrow E$

$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow (E)$

$T \rightarrow int * T$

$T \rightarrow int$

Then, Closure of $\{S' \rightarrow \bullet E\}$ is

$$\left\{ \begin{array}{l} S' \rightarrow \bullet E \\ E \rightarrow \bullet T \\ E \rightarrow \bullet T + E \\ T \rightarrow \bullet (E) \\ T \rightarrow \bullet int * T \\ T \rightarrow \bullet int \end{array} \right\}$$

GOTO function

- Goto(I, X) = closure of the set of all items
  $A \to \alpha X \bullet \beta$ where $A \to \alpha \bullet X \beta$ belongs to I

- Intuitively: Goto(I, X) set of all items that are "reachable" from the items of I once X has been "seen."

- E.g., consider $I = \{E \to T \bullet + E\}$ and compute Goto(I, +)

  Goto(I, +) = $\{E \to T + \bullet E, E \to \bullet T, E \to \bullet T + E,$
  $T \to \bullet (E), T \to \bullet \text{int} * T, T \to \bullet \text{int}\}$

Construction of SLR(1) Transition Diagram

Start with the production $S' \rightarrow \bullet S$

Create the initial state to be closure($\{S' \rightarrow \bullet S\}$)

Pick a state I

for each $A \rightarrow \alpha \bullet X \beta$ in I, find goto(I, X)

if goto(I, X) is a new state, add it to the diagram

Also, add an edge for X from state I to state goto(I, X)

Repeat until no more additions is possible

Construction of SLR(1) T.D. - Example

$S' \rightarrow .\,E$

$E \rightarrow .\,T$

$E \rightarrow .T + E$

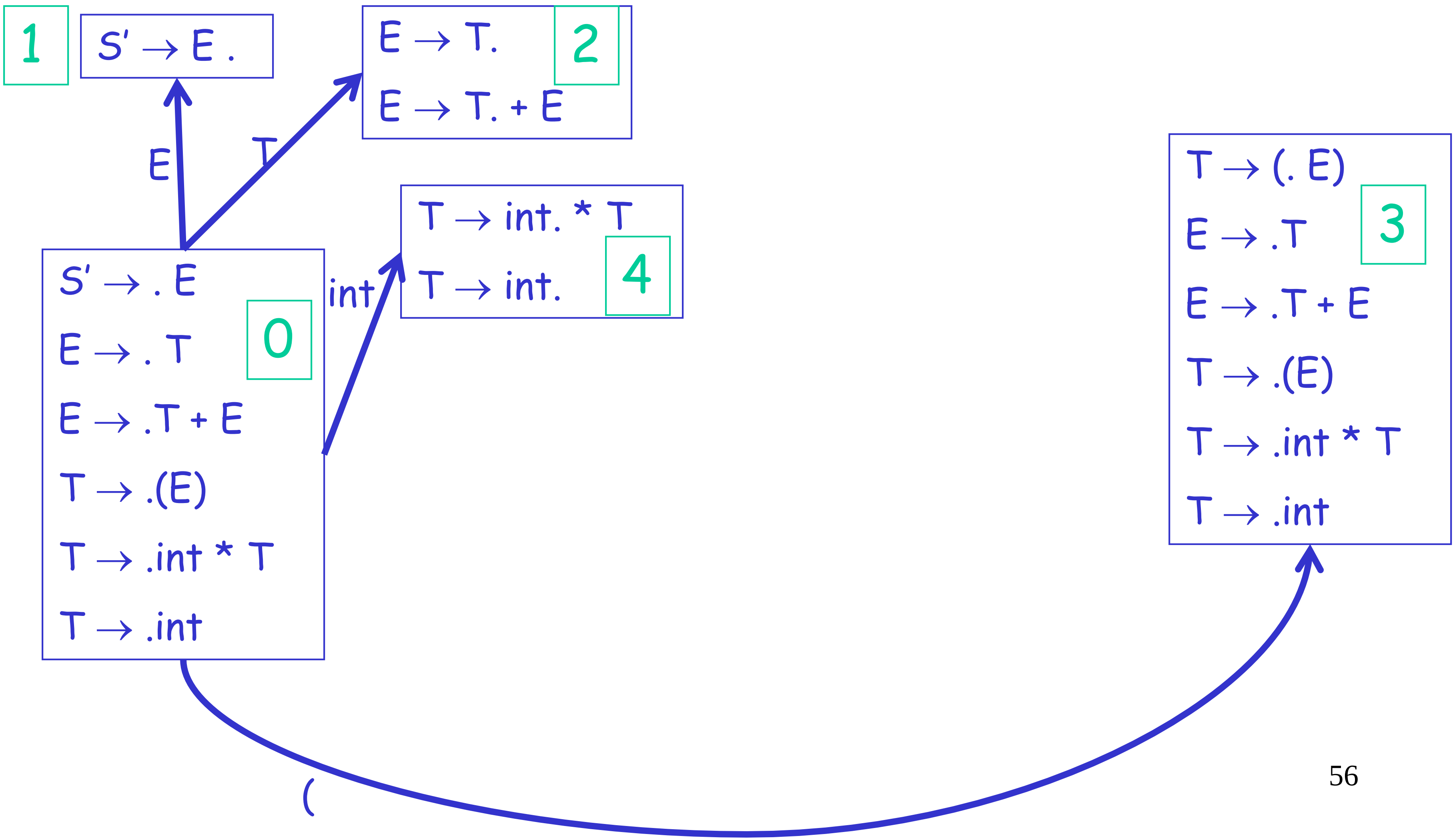
$T \rightarrow .(E)$

$T \rightarrow .int * T$

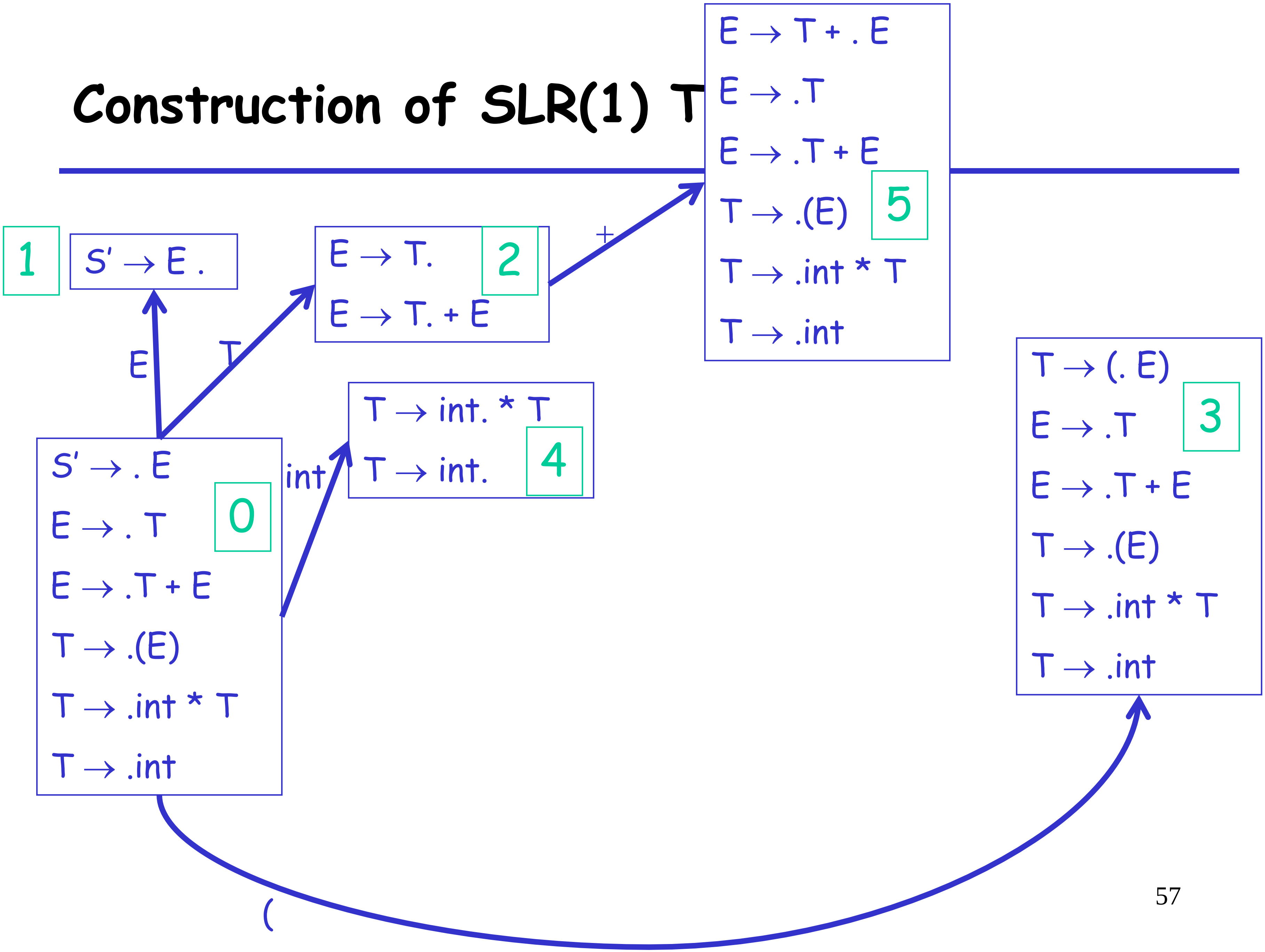
$T \rightarrow .int$

0

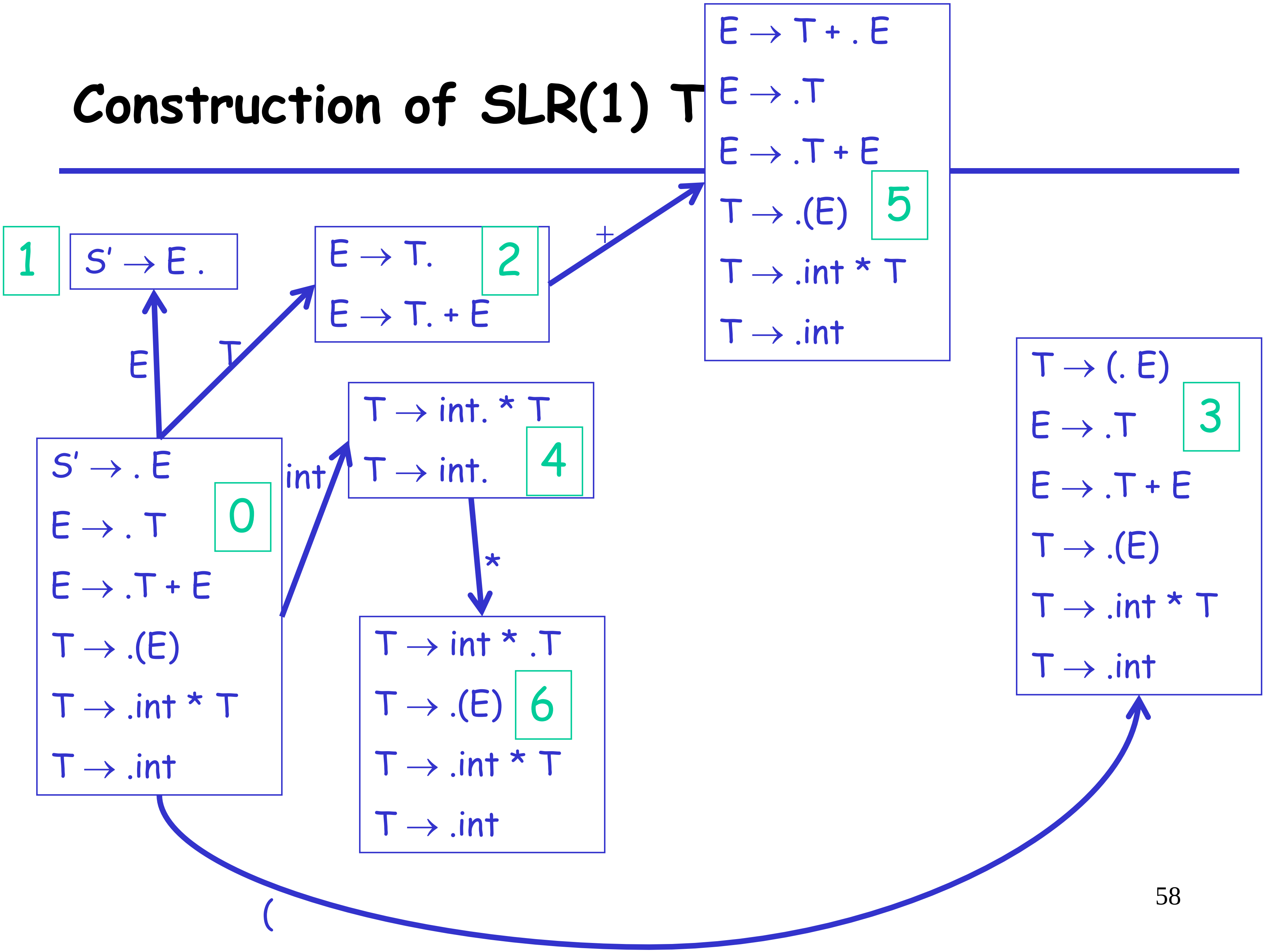
Construction of SLR(1) T.D.



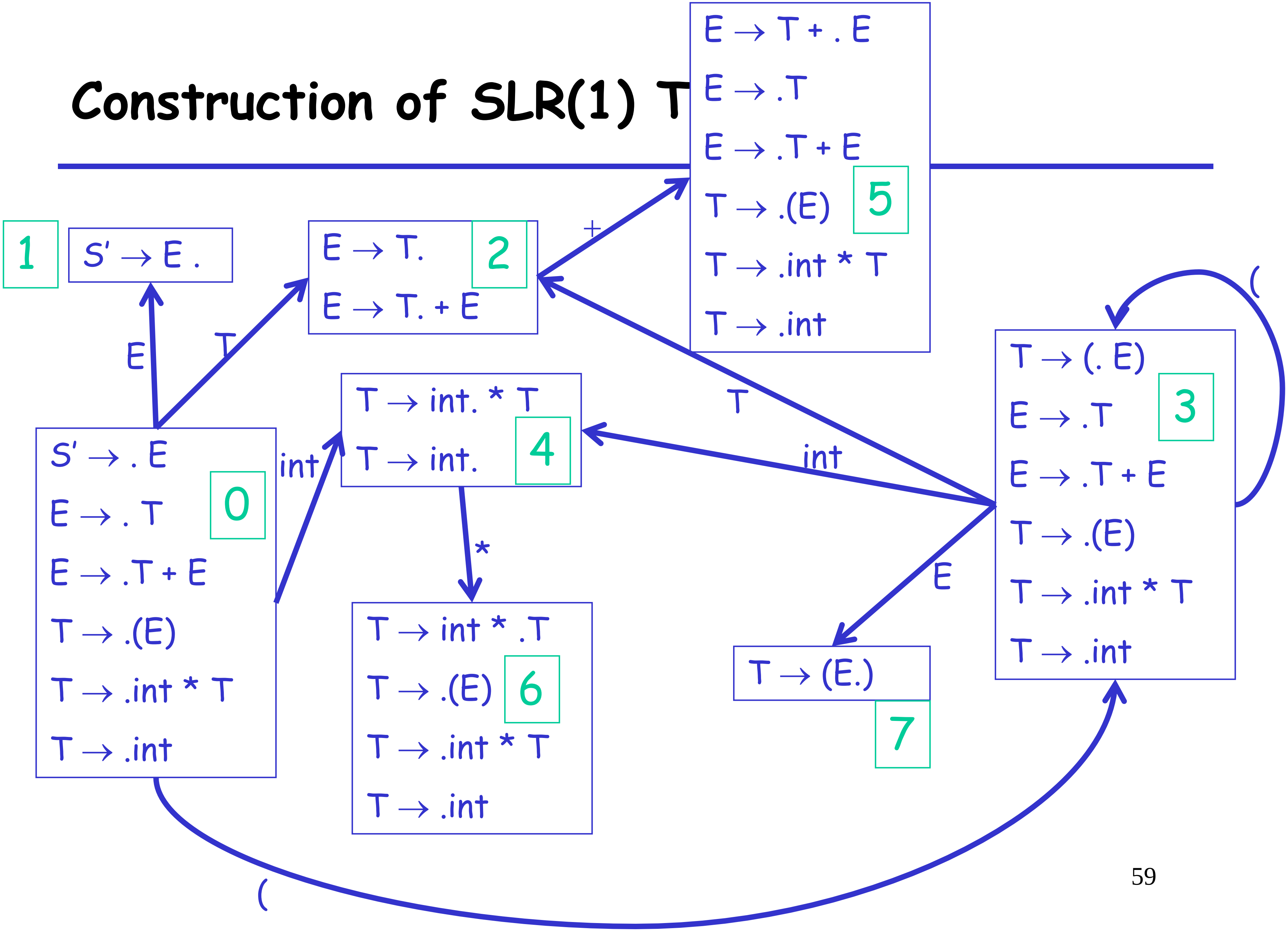
Construction of SLR(1) T



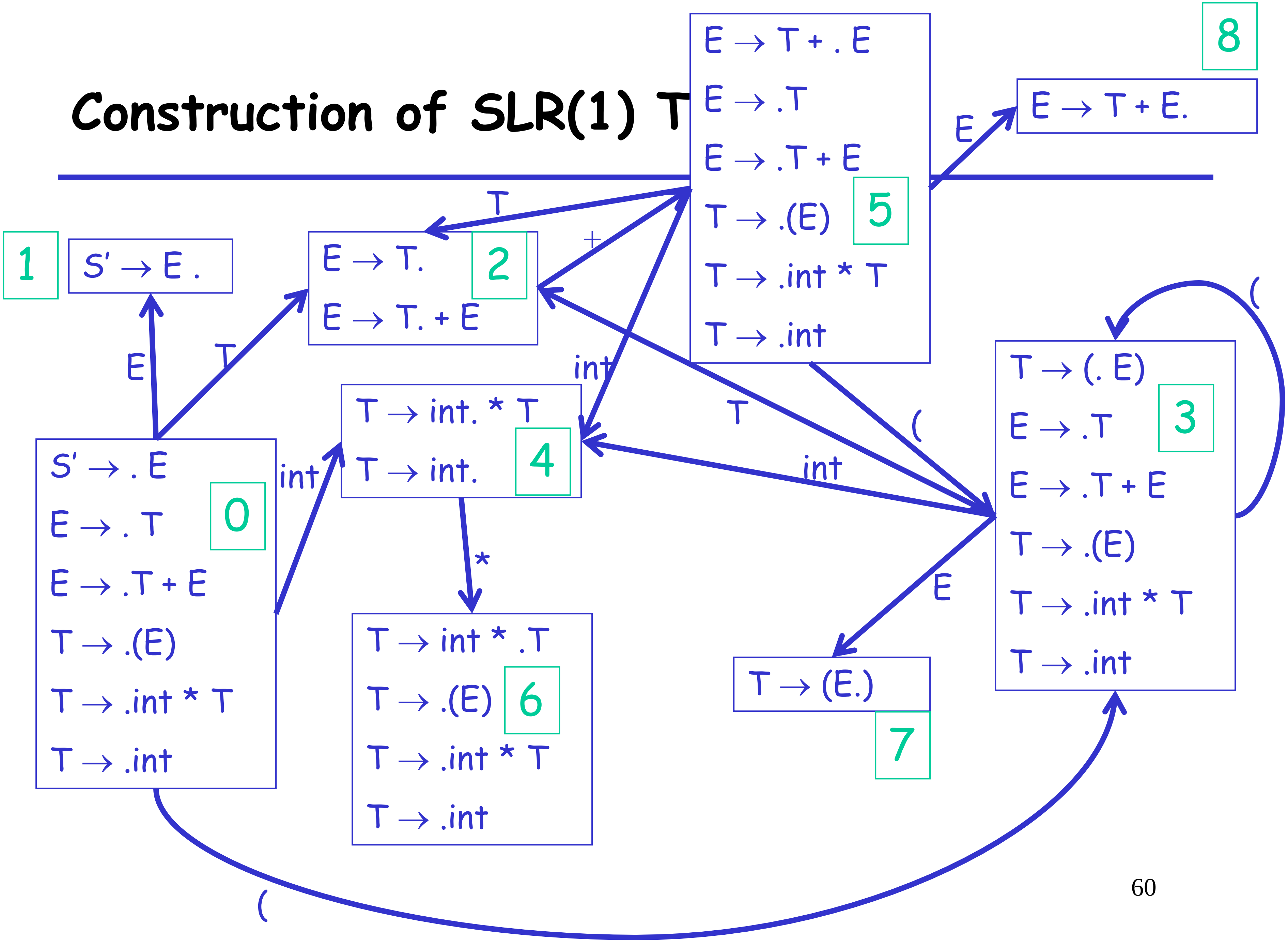
Construction of SLR(1) T



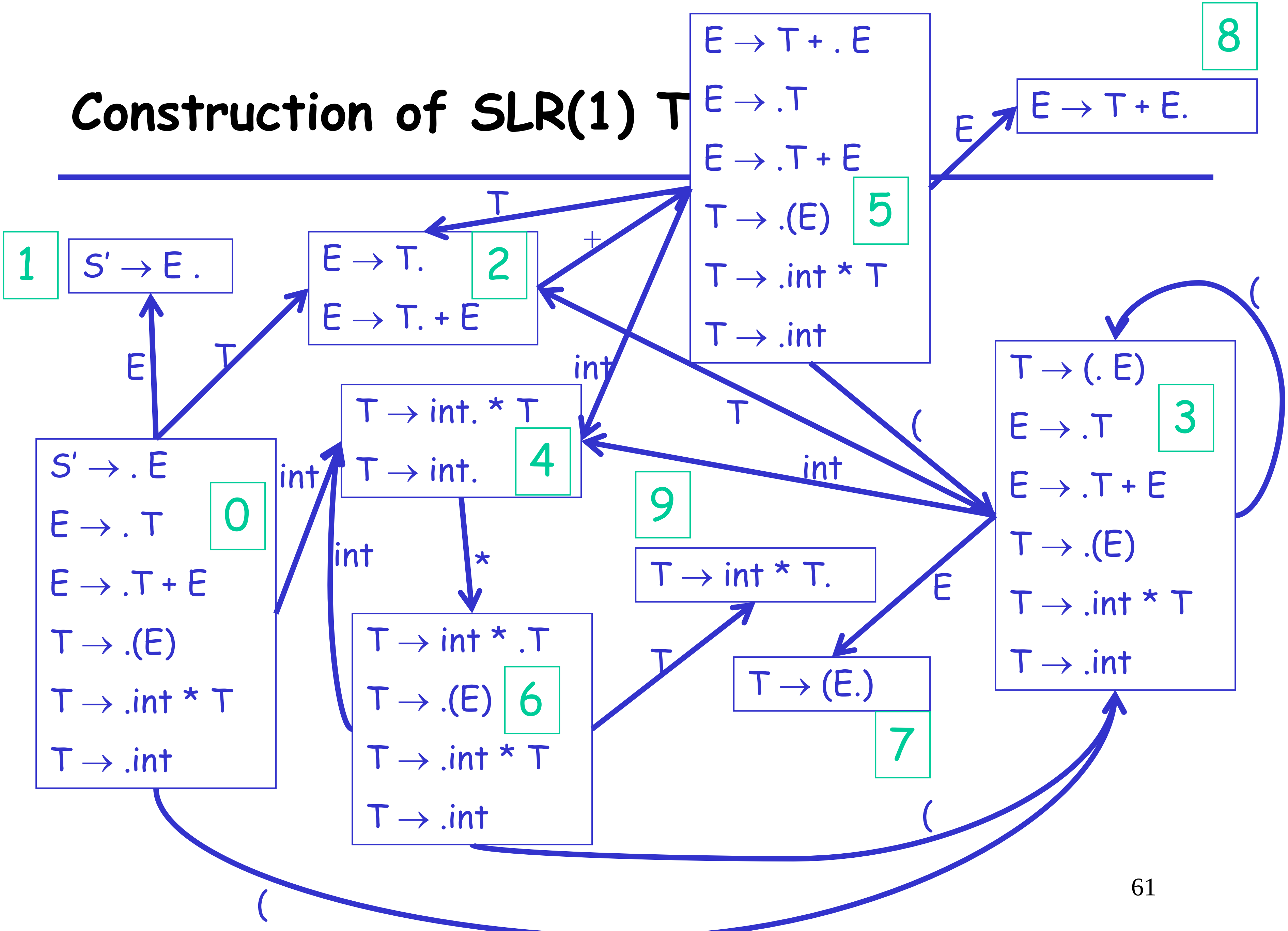
Construction of SLR(1) T



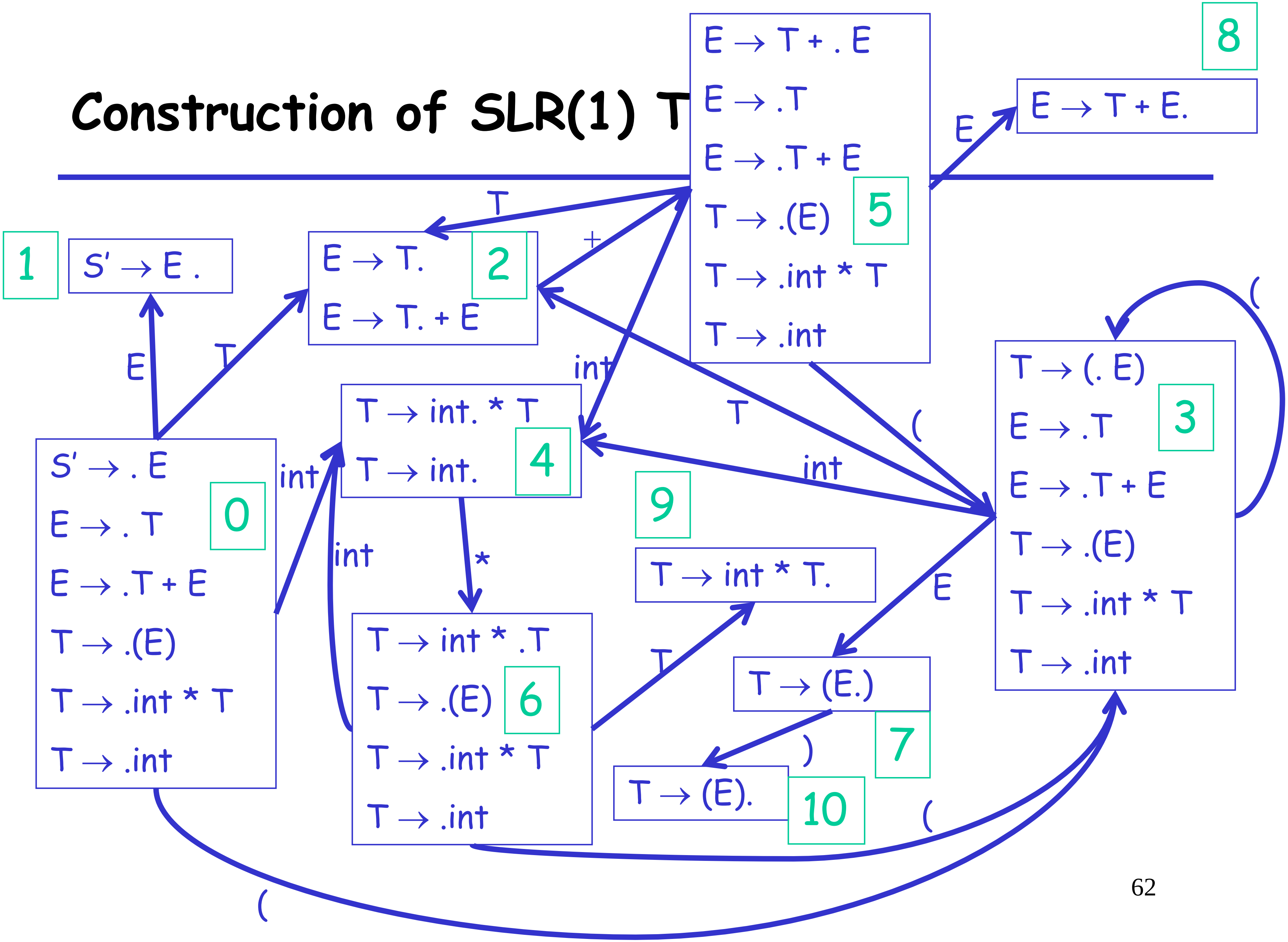
Construction of SLR(1) T



Construction of SLR(1) T



Construction of SLR(1) T



Action Table

For each state s_i and terminal a

- If s_i has item $X \to \alpha \bullet a \beta$ and $goto[i,a] = j$ then $action[i,a] = shift\ j$

- If s_i has item $X \to \alpha \bullet$ and $a \in Follow(X)$ and $X \neq S'$ then $action[i,a] = reduce\ X \to \alpha$

- If s_i has item $S' \to S \bullet$ then $action[i,\$] = accept$

- Otherwise, $action[i,a] = error$

Goto Table

For each state s_i and non-terminal A

- If s_i has item $X \rightarrow \alpha \bullet A\ \beta$ and goto[i,A] = j then action[i,A] = j

- Empty cells in Goto Table do not represent an error situation (they are never accessed!)

SLR(1) Parsing Table Example (Revisited)

1 E → T

2 E → T + E

3 T → (E)

4 T → int * T

5 T → int

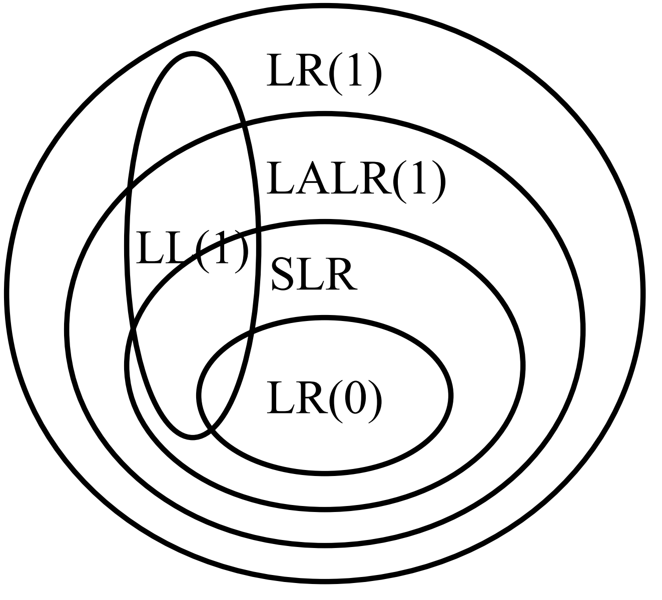
Follow (E) = {$,)}

Follow (T) = {$,), +}

	Action							Goto	
state	**int**	**+**	*****	**(**	**)**	**$**		**E**	**T**
0	S4			S3				1	2
1						acc			
2		S5			R1	R1			
3	S4			S3				7	2
4		R5	S6		R5	R5			
5	S4			S3				8	2
6	S4			S3					9
7					S10				
8					R2	R2			
9		R4			R4	R4			
10		R3			R3	R3			

LL(1) Versus LR(1) Grammars

- The class of LR grammars is a proper superset of LL(1) grammars.



Question?

For the given grammar, what is the correct series of reductions for the string: -(id + id) + id

E → E' | E' + E

E' → -E' | id | (E)

-(id + id) + id
-(id + E') + id
-(id + E) + id
-(E' + E) + id
-(E) + id
-E' + id
E' + id
E' + E'
E' + E
E

-(id + id) + id
-(E' + id) + id
-(E' + E') + id
-(E' + E) + id
-(E) + id
-E' + id
E' + id
E' + E'
E' + E
E

-(id + id) + id
-(E' + id) + id
-(E' + E') + id
-(E' + E') + E'
-(E' + E) + E'
-(E) + E'
-E' + E'
E' + E'
E' + E
E

-(id + id) + id
-(id + id) + E'
-(id + id) + E
-(E' + id) + E
-(E' + E') + E
-(E' + E) + E
-(E) + E
-E' + E
E' + E
E

Question?

For the given grammar, what is the correct shift-reduce parse for the string: id + -id

E → E' | E' + E

E' → -E' | id | (E)

```
|id +-id
id|+ -id
E'+|-id
E'+-|id
E'+-id|
E'+-E'|
E'+E'|
E'+E|
E|
```

```
|id +-id
id|+ -id
id +|-id
id +-|id
id +-id|
id +-E'|
id +E'|
id +E|
E'+E|
E|
```

```
|id +-id
|E' + -id
E'|+ -id
E' +|-id
E' + -|id
E'+-|E'
E'+|-E'
E'+|E'
E'+|E
E'|+ E
|E' +E
|E
```

```
|id +-id
id|+ -id
E'|+ -id
E'+|-id
E'+-|id
E'+-id|
E'+-E'|
E'+E'|
E'+E|
E|
```

Question?

Given the grammar at right, identify the handle for the following shift-reduce parse state: E' + -id | + -(id + id)

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

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

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

Question?

Using the DFA on slide 61, choose the next action for the given parse state

Configuration	*DFA Current State*
int * int | + int $	4

- shift
- red. T→int
- red. T→ int * T
- accept

Question?

What are the items in the initial state of the SLR(1) parsing automaton of the following grammar? Do not add extra symbol to the grammar. [Choose all that apply]

$S \to A\ (\ S\ )\ B \mid \varepsilon$
$A \to S \mid S\ B\ x \mid \varepsilon$
$B \to S\ B \mid y$

- ○ $A \to \bullet x$
- ○ $S \to \bullet A(S)B$
- ○ $A \to \bullet$
- ○ $B \to \bullet\ y$
- ○ $B \to \bullet$
- ○ $A \to \bullet S$
- ○ $A \to \bullet SBx$
- ○ $S \to \bullet$
- ○ $B \to \bullet SB$
- ○ $A \to S \bullet Bx$

Question?

Which of the followings are true for the initial state of the SLR(1) parsing automaton from the last question? [Choose all that apply]

$S \rightarrow A\ (\ S\ )\ B \mid \varepsilon$
$A \rightarrow S \mid S\ B\ x \mid \varepsilon$
$B \rightarrow S\ B \mid y$

- The state has a reduce-reduce conflict on input x.
- The state has shift-reduce conflict on transition S.
- The state has a reduce-reduce conflict on transition S.
- The state has a shift-reduce conflict on input x.
- The state has a reduce-reduce conflict on input (.

Question?

Consider the following grammar:

$$S \rightarrow A\ b \mid B\ c$$
$$A \rightarrow a\ B \mid \varepsilon$$
$$B \rightarrow b\ A \mid \varepsilon$$

This grammar is:

- ○ LL(1) but not SLR(1)
- ○ SLR(1) but not LL(1)
- ○ Not SLR(1) or LL(1)
- ○ Both LL(1) and SLR(1)