Bottom-Up Syntax Analysis

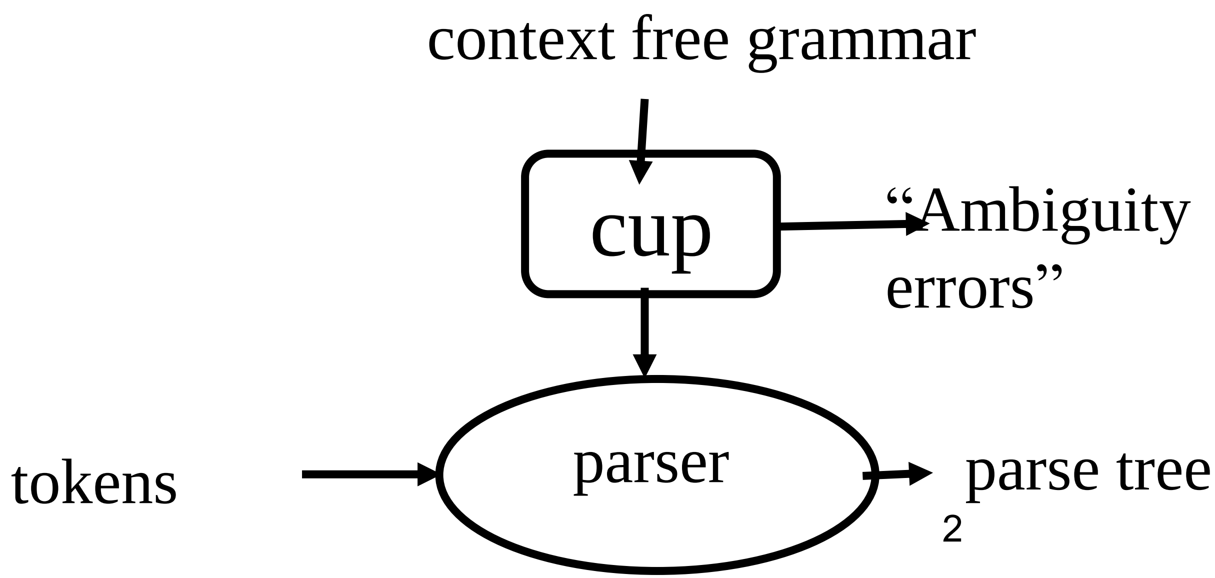
Mooly Sagiv

Textbook:Modern Compiler Design

Chapter 2.2.5 (modified)

Efficient Parsers

- Pushdown automata
- Deterministic
- Report an error as soon as the input is not a prefix of a valid program
- Not usable for all context free grammars



Kinds of Parsers

- Top-Down (Predictive Parsing) LL
  - Construct parse tree in a top-down matter
  - Find the leftmost derivation
  - For every non-terminal and token **predict** the next production
- Bottom-Up LR
  - Construct parse tree in a bottom-up manner
  - Find the rightmost derivation in a reverse order
  - For every potential right hand side and token decide when a production is found

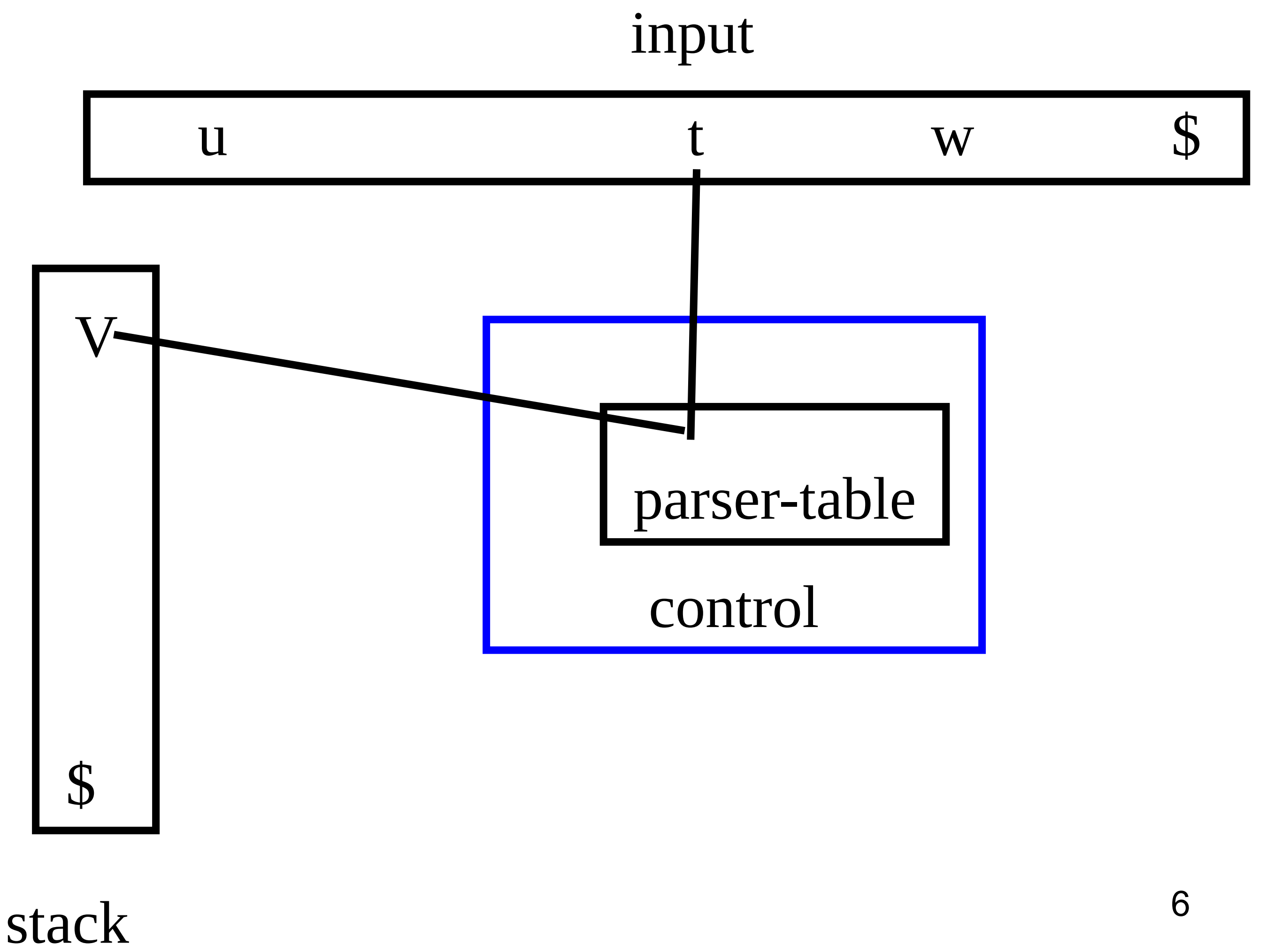
Bottom-Up Syntax Analysis

- Input
  - A context free grammar
  - A stream of tokens
- Output
  - A syntax tree or error
- Method
  - Construct parse tree in a bottom-up manner
  - Find the rightmost derivation in (reversed order)
  - For every potential right hand side and token decide when a production is found
  - Report an error as soon as the input is not a prefix of valid program

Plan

- Pushdown automata
- Bottom-up parsing (informal)
- Non-deterministic bottom-up parsing
- Deterministic bottom-up parsing
- Interesting non LR grammars

Pushdown Automaton



Informal Example(1)

$S \to E\$ \quad E \to T \mid E + T \quad T \to i \mid (E)$

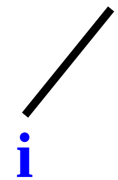
stack **tree** **input**

stack	tree	input
$		i + i $

shift

stack **tree** **input**

stack	tree	input
i$		+ i $



Informal Example(2)

$S \rightarrow E\$ \quad E \rightarrow T \mid E + T \qquad T \rightarrow i \mid (E)$

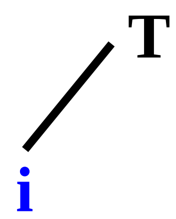
stack	tree	input
i$	i	+ i $

reduce $T \rightarrow i$

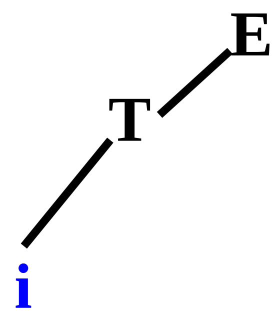
stack	tree	input
T $	T i	+ i $

Informal Example(3)

S → E$ E → T | E + T T → i | (E)

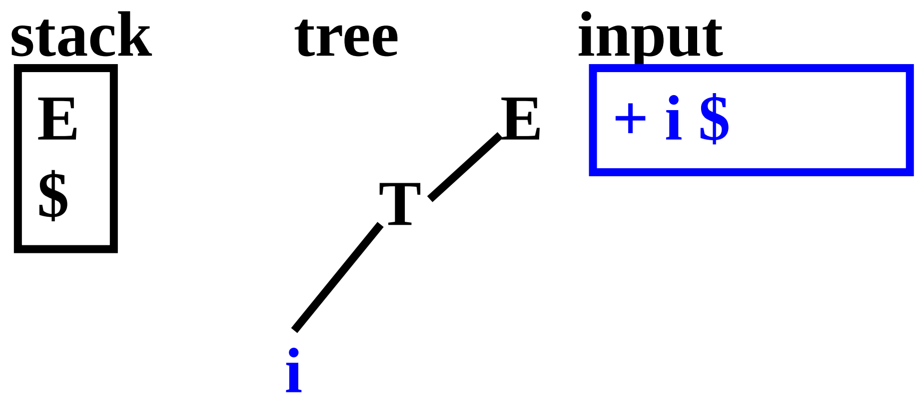
stack	**tree**	**input**
T $		+ i $

reduce E → T

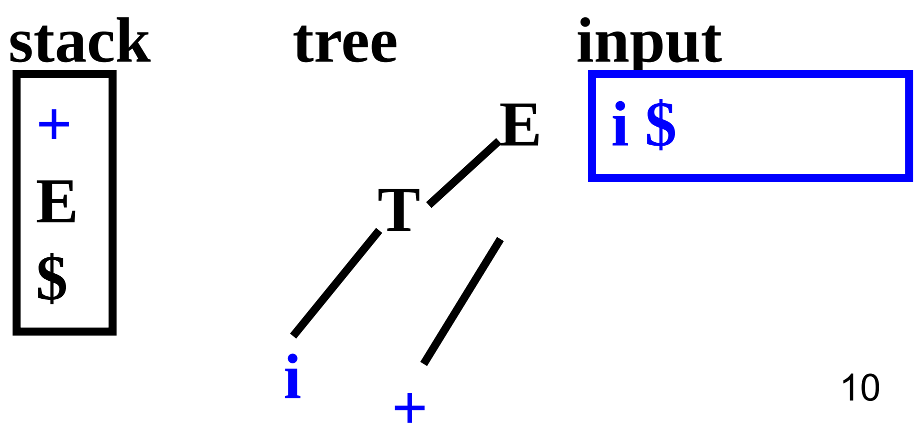
stack	**tree**	**input**
E $		+ i $

Informal Example(4)

$S \rightarrow E\$ \quad E \rightarrow T \mid E + T \quad T \rightarrow i \mid (E)$



shift

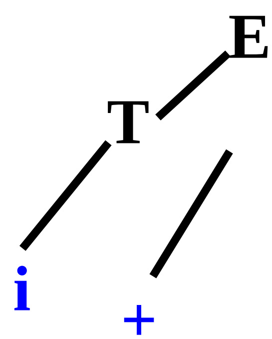


Informal Example(5)

$S \rightarrow E\$ \quad E \rightarrow T \mid E + T \quad T \rightarrow i \mid (E)$

stack **tree** **input**

stack
+
E
$

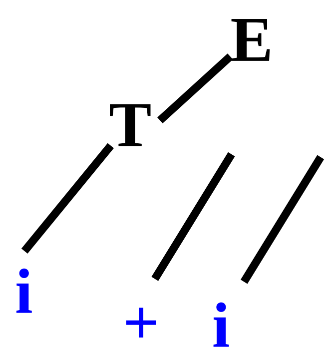


input
i $

shift

stack **tree** **input**

stack
i+
E
$



input
$

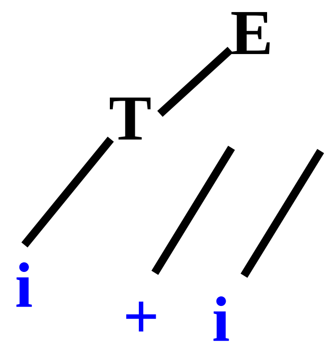
Informal Example(6)

$S \to E\$ \quad E \to T \mid E + T \quad T \to i \mid (E)$

stack **tree** **input**

Stack:

i+
E
$



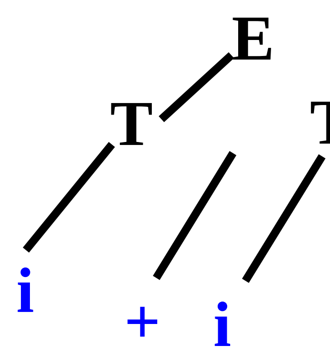
Input: $

reduce $T \to i$

stack **tree** **input**

Stack:

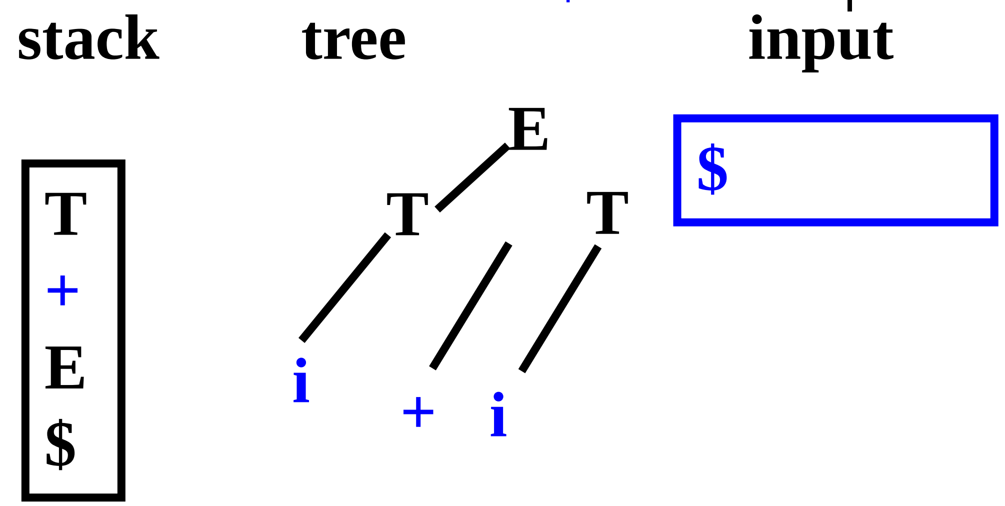
T
+
E
$



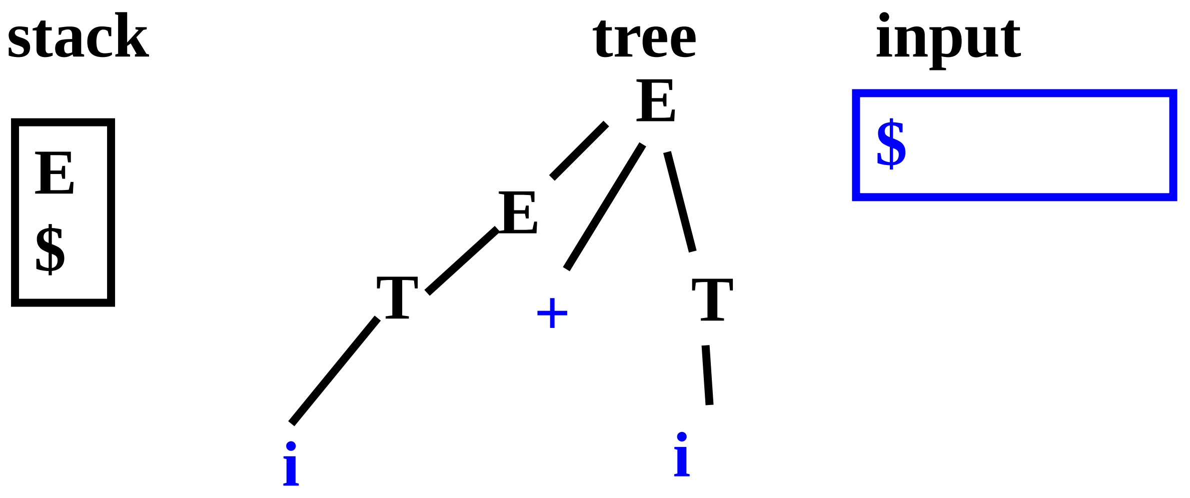
Input: $

Informal Example(7)

$S \rightarrow E\$ \quad E \rightarrow T \mid E + T \quad T \rightarrow i \mid (E)$



reduce $E \rightarrow E + T$

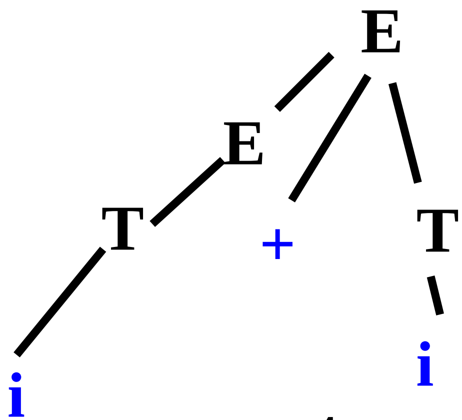


Informal Example(8)

S → E$ E → T | E + T T → i | (E)

stack **tree** **input**

stack
E
$

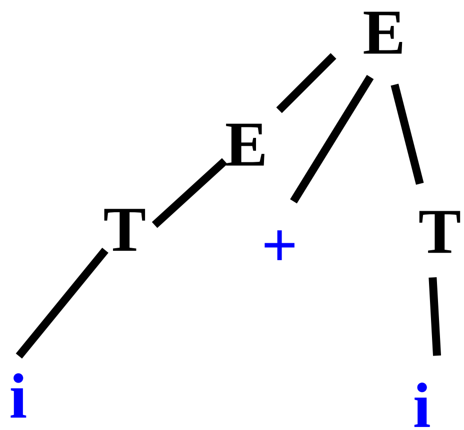


input
$

shift

stack **tree** **input**

stack
$
E
$



input

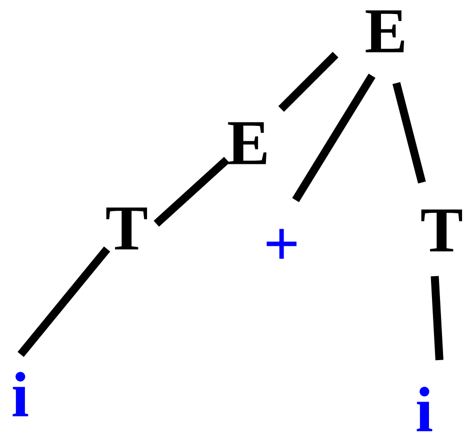
Informal Example(9)

S → E$ E → T | E + T T → i | (E)

stack

$
E
$

tree



input

$

reduce S → E $

Informal Example

reduce S $\rightarrow$ E $

reduce E $\rightarrow$ E + T

reduce T $\rightarrow$ i

reduce E $\rightarrow$ T

reduce T $\rightarrow$ i

The Problem

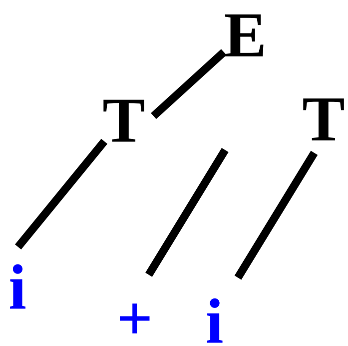
- Deciding between shift and reduce

Informal Example(7)

S → E$ E → T | E + T T → i | (E)

stack **tree** **input**

T
+
E
$

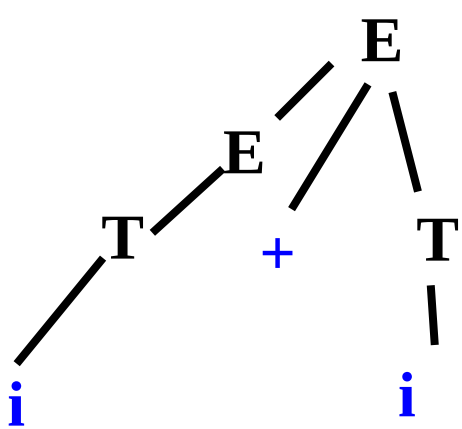


$

reduce E → E + T

stack **tree** **input**

E
$



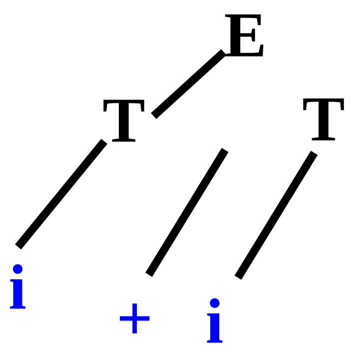
$

Informal Example(7’)

S → E$ E → T | E + T   T → i | (E)

stack | **tree** | **input**

T
+
E
$

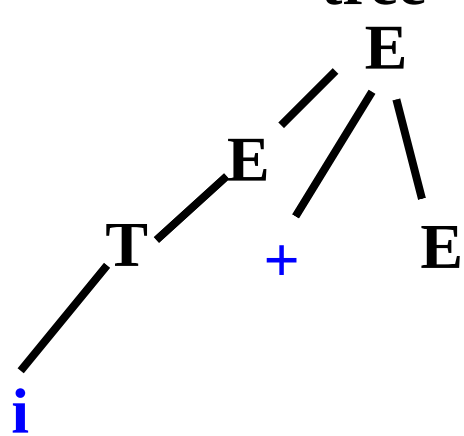


$

reduce E → T

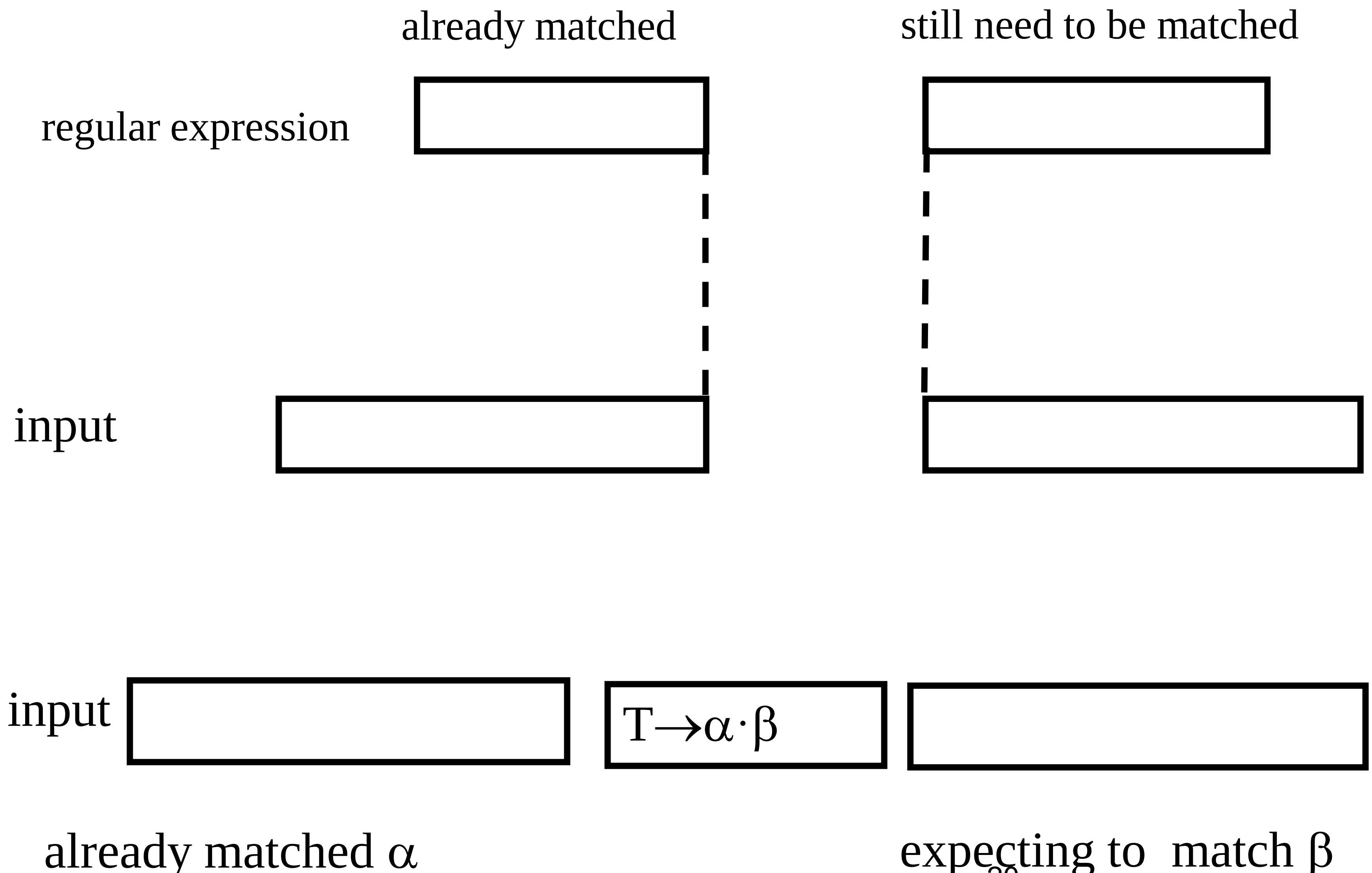
stack | **tree** | **input**

E
+
E
$



$

Bottom-UP LR(0) Items



S → E$

E → T

E → E + T

T → i

T → (E)

LR(0) items	(	)	i	+	$	T	E	ε
1: S → •E$							2	4, 6
2: S → E • $					s3			
3: S → E $ •								r
4: E → • T						5		10, 12
5: E → T •								r
6: E → • E + T							7	4, 6
7: E → E • + T				s8				
8: E → E + • T						9		10, 12
9: E → E + T •								r
10: T → • i			s11					
11: T → i •								r
12: T → • (E)	s13							
13: T → (• E)							14	4, 6
14: T → (E •)		s15						
15: T → (E) •								r

Formal Example(1)

S → E$ E → T | E + T   T → i | (E)

stack

1: S → •E$

input

i + i $

ε-move 6

stack

6: E → •E+T

1: S → •E$

input

i + i $

Formal Example(2)

S → E$ E → T | E + T   T → i | (E)

stack

input

6: E → •E+T
1: S → •E$

i + i $

ε-move 4

stack

input

4: E → •T
6: E → •E+T
1: S → •E$

i + i $

Formal Example(3)

S → E$ E → T | E + T T → i | (E)

stack	**input**
4: E → •T **6: E → •E+T** **1: S → •E$**	**i + i $**

ε-move 10

stack	**input**
10: T → • i **4: E → •T** **6: E → •E+T** **1: S → •E$**	**i + i $**

Formal Example(4)

$S \rightarrow E\$ \quad E \rightarrow T \mid E + T \quad T \rightarrow i \mid (E)$

stack

10: T → ● i
4: E → ●T
6: E → ●E+T
1: S → ●E$

input

i + i $

stack

11: T → i ●
10: T → ● i
4: E → ●T
6: E → ●E+T
1: S → ●E$

input

+ i $

shift 11

Formal Example(5)

$S \rightarrow E\$ \quad E \rightarrow T \mid E + T \quad T \rightarrow i \mid (E)$

stack **input**

stack
11: T → i •
10: T → • i
4: E → •T
6: E → •E+T
1: S → •E$

+ i $

reduce T → i

stack **input**

stack
5: E → T •
4: E → •T
6: E → •E+T
1: S → •E$

+ i $

Formal Example(6)

$S \rightarrow E\$ \quad E \rightarrow T \mid E + T \quad T \rightarrow i \mid (E)$

stack **input**

5: E → T •
4: E → •T
6: E → •E+T
1: S → •E$

+ i $

reduce E → T

stack **input**

7: E → E • +T
6: E → •E+T
1: S → •E$

+ i $

Formal Example(7)

S → E$ E → T | E + T   T → i | (E)

stack	input
7: E → E • +T 6: E → •E+T 1: S → •E$	+ i $

shift 8

stack	input
8: E → E + • T 7: E → E • +T 6: E → •E+T 1: S → •E$	i $

Formal Example(8)

$S \to E\$ \quad E \to T \mid E + T \quad T \to i \mid (E)$

stack **input**

8: $E \to E + \bullet T$
7: $E \to E \bullet +T$
6: $E \to \bullet E+T$
1: $S \to \bullet E\$$

input: **i $**

stack **input** **ε-move 10**

10: $T \to \bullet i$
8: $E \to E + \bullet T$
7: $E \to E \bullet +T$
6: $E \to \bullet E+T$
1: $S \to \bullet E\$$

input: **i $**

Formal Example(9)

S → E$ E → T | E + T   T → i | (E)

stack **input**

10: T → • i
8: E → E + • T
7: E → E • +T
6: E → •E+T
1: S → •E$

i $

shift 11

stack **input**

11: T → i •
10: T → • i
8: E → E + • T
7: E → E • +T
6: E → •E+T
1: S → •E$

$

Formal Example(10)

$S \to E\$ \quad E \to T \mid E + T \quad T \to i \mid (E)$

stack

11: $T \to i \bullet$
10: $T \to \bullet i$
8: $E \to E + \bullet T$
7: $E \to E \bullet +T$
6: $E \to \bullet E+T$
1: $S \to \bullet E\$$

input

$

reduce $T \to i$

stack

9: $E \to E + T \bullet$
8: $E \to E + \bullet T$
7: $E \to E \bullet +T$
6: $E \to \bullet E+T$
1: $S \to \bullet E\$$

input

$

Formal Example(11)

$S \rightarrow E\$ \quad E \rightarrow T \mid E + T \quad T \rightarrow i \mid (E)$

stack

9: E → E + T •
8: E → E + • T
7: E → E • +T
6: E → •E+T
1: S → •E$

input

$

reduce E → E + T

stack

2: S → E • $
1: S → •E$

input

$

Formal Example(12)

$$S \rightarrow E\$ \quad E \rightarrow T \mid E + T \quad T \rightarrow i \mid (E)$$

stack **input**

stack	input
2: S → E • $	$
1: S → •E$	

shift 3

stack **input**

stack	input
3: S → E $ •	
2: S → E • $	
1: S → •E$	

Formal Example(13)

$S \rightarrow E\$ \quad E \rightarrow T \mid E + T \quad T \rightarrow i \mid (E)$

stack **input**

3: S → E $ •
2: S → E • $
1: S → •E$

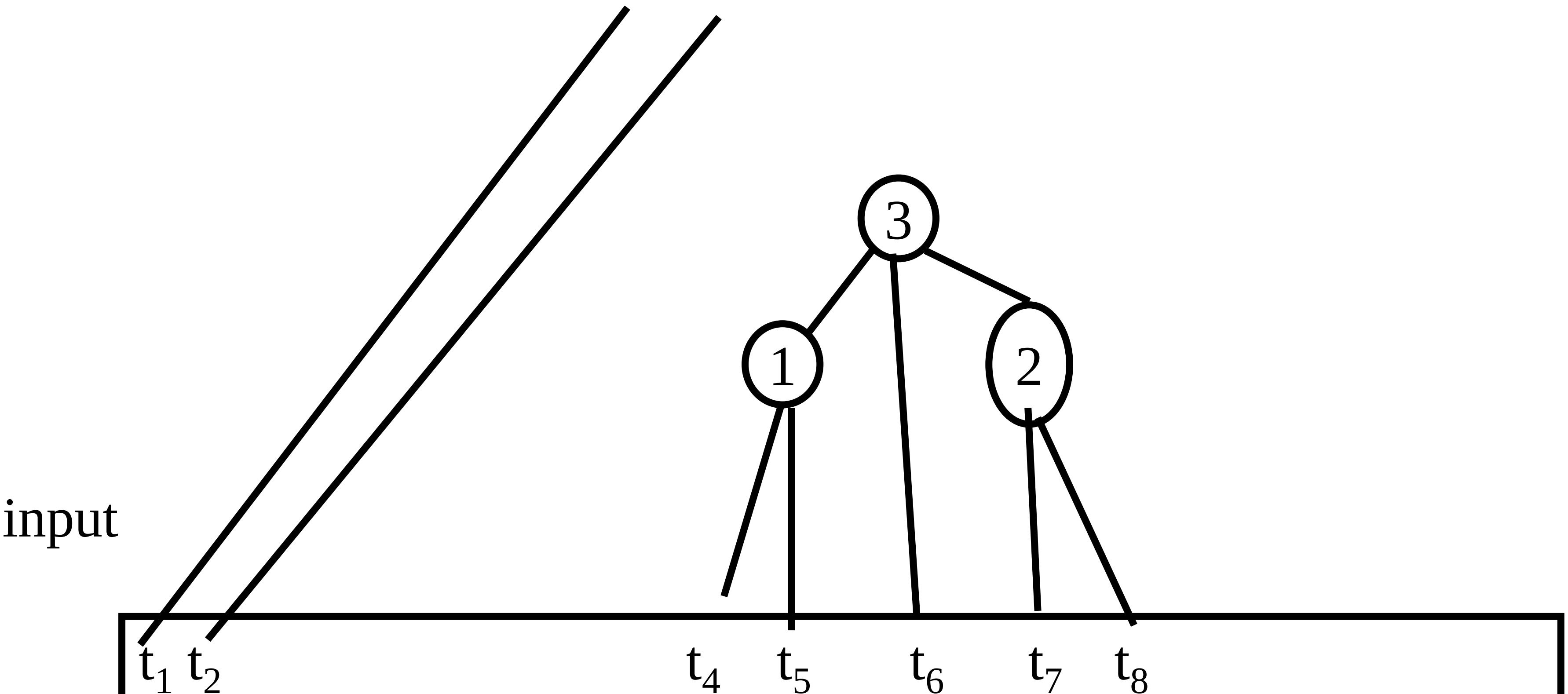
reduce S → E$

But how can this be done efficiently?

Deterministic Pushdown Automaton

Handles

- Identify the leftmost node (nonterminal) that has not been constructed but all whose children have been constructed



Identifying Handles

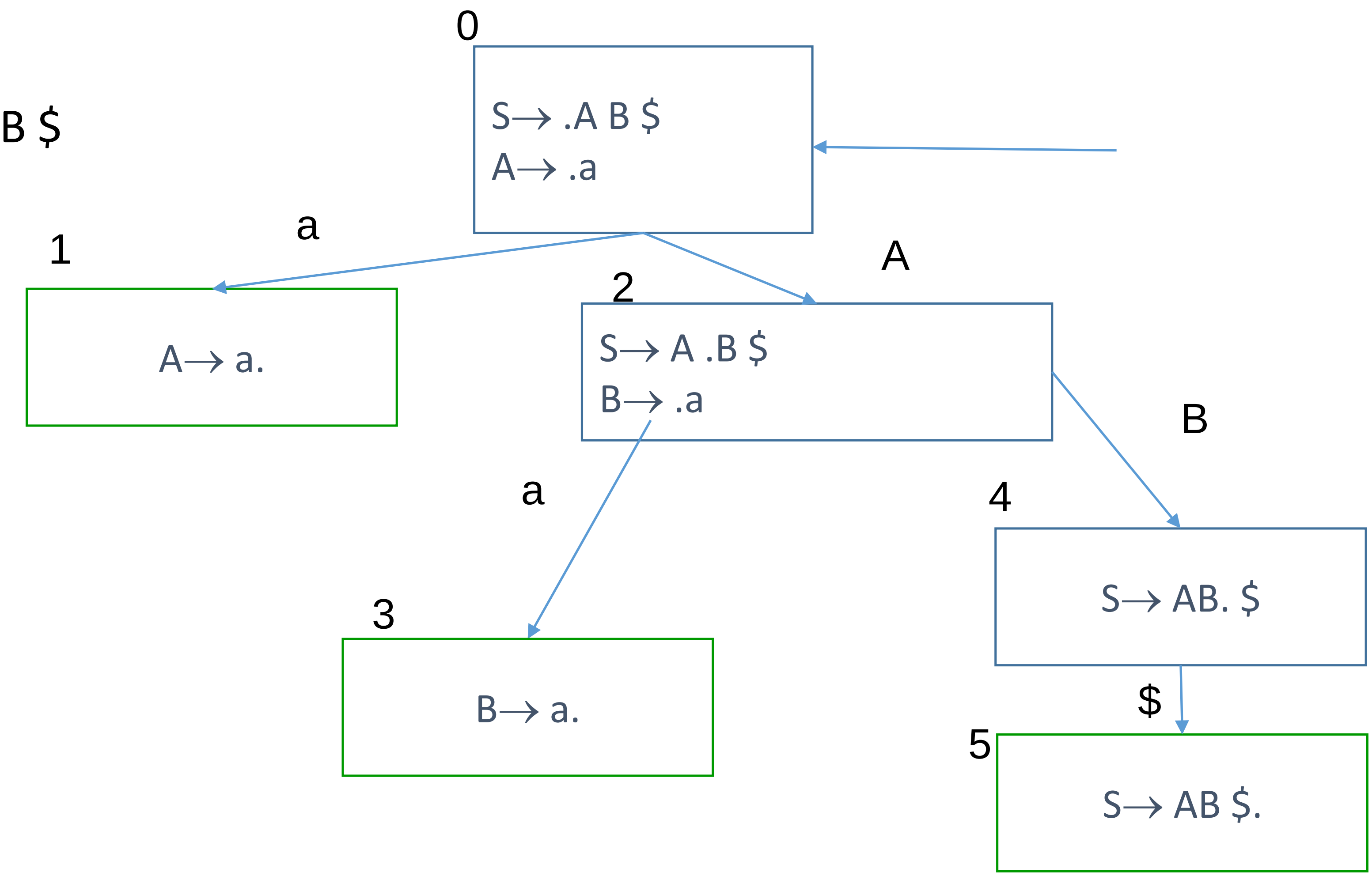
- The language of stack symbols $(T \cup N)^*$ is regular
- Create a deterministic finite state automaton over grammar symbols
  - Sets of LR(0) items $A \rightarrow \alpha \bullet \beta$
    - Identified α
    - Expecting β
- Use automaton to build parser tables
  - reduce For items $A \rightarrow \alpha \bullet$ on every token
  - shift For items $A \rightarrow \alpha \bullet t \beta$ on token t
  - goto For items $A \rightarrow \alpha \bullet X \beta$ on nonterminal X

Identifying Handles

- Create a deterministic finite state automaton over grammar symbols
  - Sets of LR(0) items
- Use automaton to build parser tables
  - reduce For items $A \rightarrow \alpha \bullet$ on every token
  - shift For items $A \rightarrow \alpha \bullet t \beta$ on token t
  - goto For items $A \rightarrow \alpha \bullet X \beta$ on nonterminal X
- When conflicts occur the grammar is not LR(0)
- When no conflicts occur use a DPDA which pushes states on the stack

A Trivial Example

- S → A B $
- A → a
- B → a



Control Table Trivial Example

state	terminal			nonterminal		
	a	$	other	S	A	B
0 S→ .A B $ A→ .a	shift 1	err	err		2	
1 A→ a.	reduce A→a					
2 S→ A .B $ B→ .a	shift 3	err	err			4
3 B→ a.	reduce B→a					
4 S→ AB. $	err	shift 5	err			
5 S→ AB $.	accept					

Parsing aa

state	terminal			nonterminal		
	a	$	other	S	A	B
0	s1	err	err		2	
1	reduce A→a					
2	s3	err	err			4
3	reduce B→a					
4	err	s5	err			
5	accept					

stack	input	action
0($)	aa $	shift 1
1(a) 0($)	a $	reduce A→a
0($)	a $	A
2(A) 0($)	a $	shift 3

Parsing aa (Cont)

state	terminal			Nonterminal		
	a	$	other	S	A	B
0	s1	err	err		2	
1	reduce A→a					
2	s3	err	err			4
3	reduce B→a					
4	err	s5	err			
5	accept					

stack	input	action
3(a) 2(A) 0($)	$	reduce B→a
4(B) 2(A) 0($)	$	shift 5
5($) 4(B) 2(A) 0($)		accept

The Rightmost Derivation in Reverse Order

1. S → BA$
2. B→a
3. A→a

Parsing ab

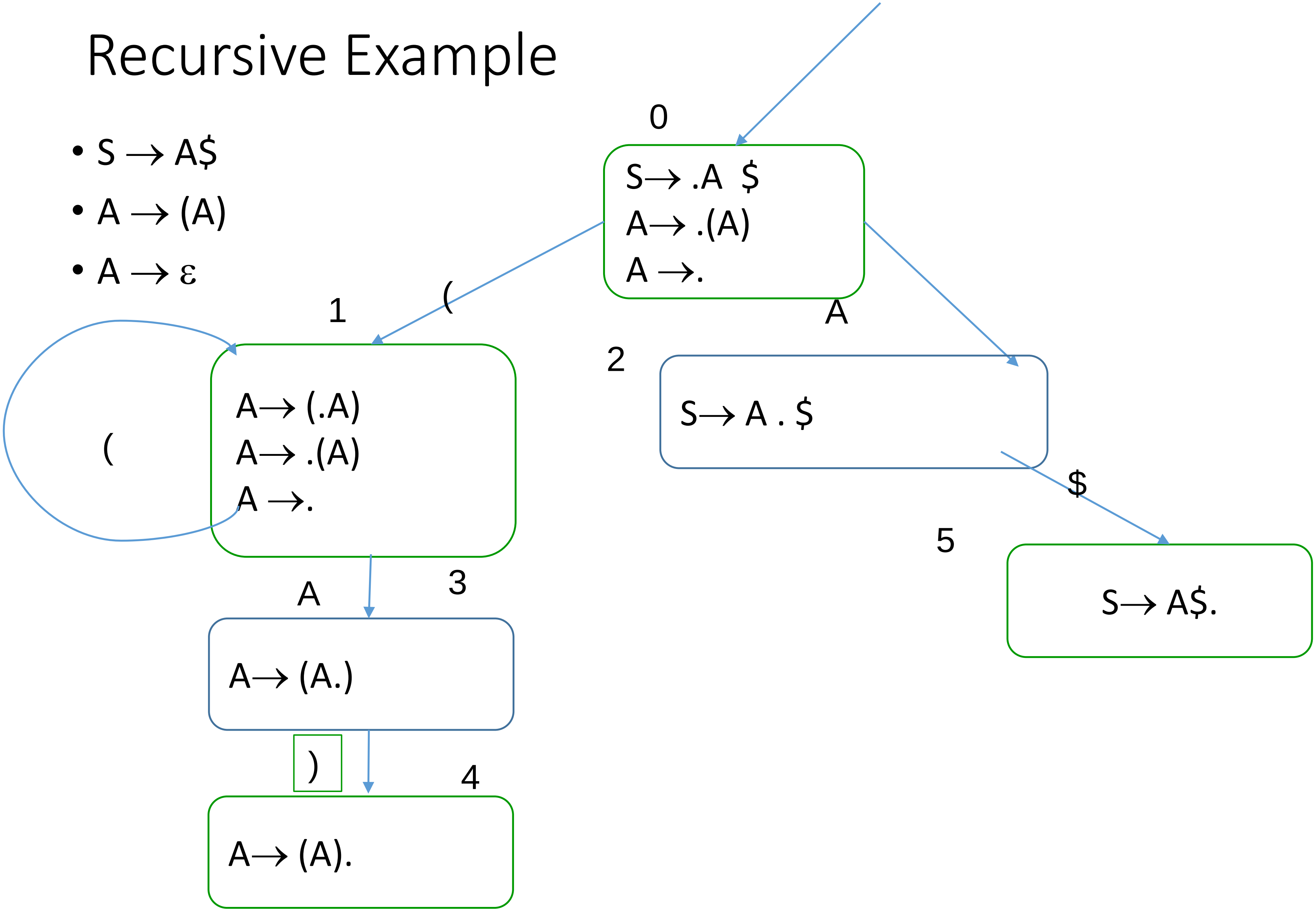
state	terminal			nonterminal		
	a	$	other	S	A	B
0	s1	err	err		2	
1	reduce A→a					
2	s3	err	err			4
3	reduce B→a					
4	err	s5	err			
5	accept					

stack	input	action
0($)	ab $	shift 1
1(a) 0($)	b$	reduce A→a
2(A) 0($)	b$	err

Does this satisfy the valid prefix property?

Recursive Example

- $S \rightarrow A\$$
- $A \rightarrow (A)$
- $A \rightarrow \varepsilon$



Control Table Recursive Example

state	terminal				nonterminal	
	(	)	$	other	S	A
0 S→ .A$ A→ .(A) A→.	shift 1/ reduce A→	err	err	err		2
1 A→ (.A) A→ .(A) A →.	shift 1/ reduce A→	reduce A→				
2 S→ A .$	shift 5	err				
3 A→ (A.)	err	shift 4				
4 A→ (A).	err	err	shift 5			
5 S→ A$.	accept					

Resolving Conflicts using one token SLR(1)

- For every token t $\in$ follow(A) and for every item A → α. $\in$S
  - reduce A → α in S
- CUP implements more sophisticated mechanism

Trivial Example Follow

- $S \rightarrow A\ B\ \$$
- $A \rightarrow a$
- $B \rightarrow a$

Follow(S) = {}

Follow(A) = {a}

Follow(B) = {$}

Control Table Trivial Example with Follow

state	terminal			nonterminal		
	a	$	other	S	A	B
0 S→ .A B $ A→ .a	shift 1	err	err		2	
1 A→ a.	reduce A→a	err	err			
2 S→ A .B $ B→ .a	shift 3	err	err			4
3 B→ a.	err	reduce B→a	err			
4 S→ AB. $	err	shift 5	err			
5 S→ AB $.	accept					

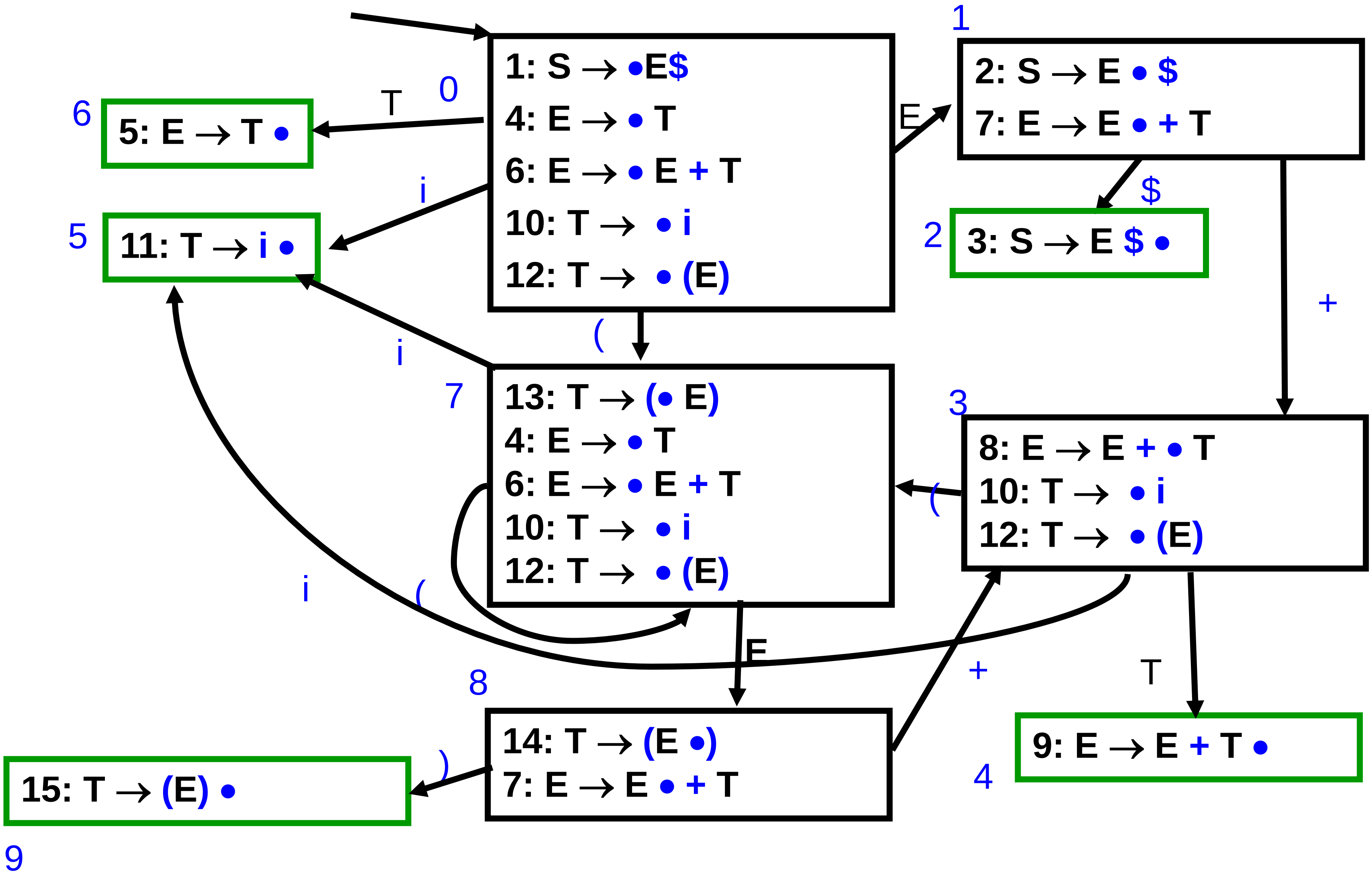
Recursive Example

- $S \rightarrow A\$$
- $A \rightarrow (A)$
- $A \rightarrow \varepsilon$

Follow(A) = {$,)}

Control Table Recursive Example with Follow

state	terminal				nonterminal	
	(	)	$	other	S	A
0 S→ .A$ A→ .(A) A→.	shift 1/ ~~reduce A→~~	err	err	err		2
1 A→ (.A) A→ .(A) A →.	shift 1/ ~~reduce A→~~	reduce A→				
2 S→ A .$	shift 5	err				
3 A→ (A.)	err	shift 4				
4 A→ (A).	err	err	shift 5			
5 S→ A$.	accept					



0
1: S → •E$
4: E → • T
6: E → • E + T
10: T → • i
12: T → • (E)
1
2: S → E • $
7: E → E • + T
2
3: S → E $ •
3
8: E → E + • T
10: T → • i
12: T → • (E)
4
9: E → E + T •
5
11: T → i •
6
5: E → T •
7
13: T → (• E)
4: E → • T
6: E → • E + T
10: T → • i
12: T → • (E)
8
14: T → (E •)
7: E → E • + T
9
15: T → (E) •
T
E
i
$
(
+
)

Example Control Table

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce $E \rightarrow E+T$						
5	reduce $T \rightarrow i$						
6	reduce $E \rightarrow T$						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce $T \rightarrow (E)$						

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

0($)

input

i + i $

shift 5

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

5 (i)
0 ($)

input

+ i $

reduce T → i

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

6 (T)
0 ($)

input

+ i $

reduce E → T

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

1(E)
0 ($)

input

+ i $

shift 3

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

3 (+)

1(E)

0 ($)

input

i $

shift 5

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

5 (i)
3 (+)
1(E)
0($)

input

$

reduce T → i

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

4 (T)

3 (+)

1(E)

0($)

input

$

reduce E → E + T

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack	**input**	
1 (E) **0 ($)**	**$**	**shift 2**

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

2 ($)

1 (E)

0 ($)

input

accept

The Rightmost Derivation in Reverse Order

1. S → E$
2. E → E + T
3. T→ I
4. E → T
5. T → i

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

0($)

input

((i) $

shift 7

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

7(()
0($)

input

(i) $

shift 7

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

7 (()
7(()
0($)

input

i) $

shift 5

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

5 (i)

7 (()

7(()

0($)

input

) $

reduce T → i

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

6 (T)
7 (()
7(()
0($)

input

) $

reduce E →T

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

8 (E)
7 (()
7(()
0($)

input

) $

shift 9

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

9 ())
8 (E)
7 (()
7(()
0($)

input

$

reduce T → (E)

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

6 (T)
7(()
0($)

input

$

reduce E → T

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

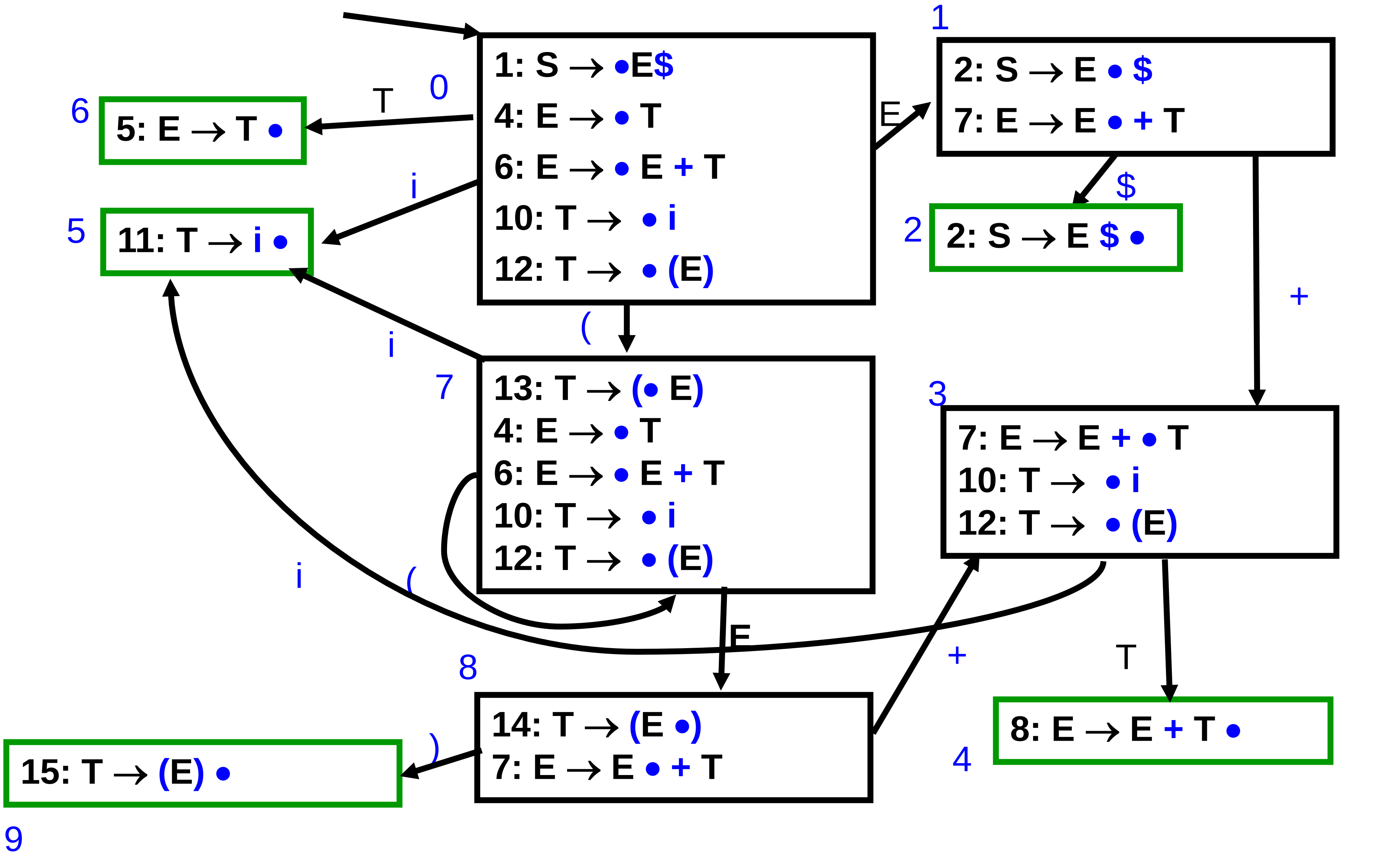
stack

8 (E)
7(()
0($)

input

$

err



0
1: S → •E$
4: E → • T
6: E → • E + T
10: T → • i
12: T → • (E)
1
2: S → E • $
7: E → E • + T
6
5: E → T •
5
11: T → i •
2
2: S → E $ •
7
13: T → (• E)
4: E → • T
6: E → • E + T
10: T → • i
12: T → • (E)
3
7: E → E + • T
10: T → • i
12: T → • (E)
8
14: T → (E •)
7: E → E • + T
4
8: E → E + T •
9
15: T → (E) •
T
E
i
$
+
(
i
i
(
E
+
T
)

Constructing LR(0) parsing table

- Add a production $S' \rightarrow S\$$
- Construct a deterministic finite automaton accepting "valid stack symbols"
- States are set of items $A \rightarrow \alpha \bullet \beta$
  - The states of the automaton becomes the states of parsing-table
  - Determine **shift** operations
  - Determine **goto** operations
  - Determine **reduce** operations

Filling Parsing Table

- A state s_i
- reduce $A \rightarrow \alpha$
  - $A \rightarrow \alpha \bullet \in s_i$
- Shift on t
  - $A \rightarrow \alpha \bullet t \, \beta \in s_i$
- Goto(s_i, X) = s_j
  - $A \rightarrow \alpha \bullet X \, \beta \in s_i$
  - $\delta(s_i, X) = s_j$
- When conflicts occurs the grammar is not LR(0)

Example Control Table

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

Example Non LR(0) Grammar

S → E$

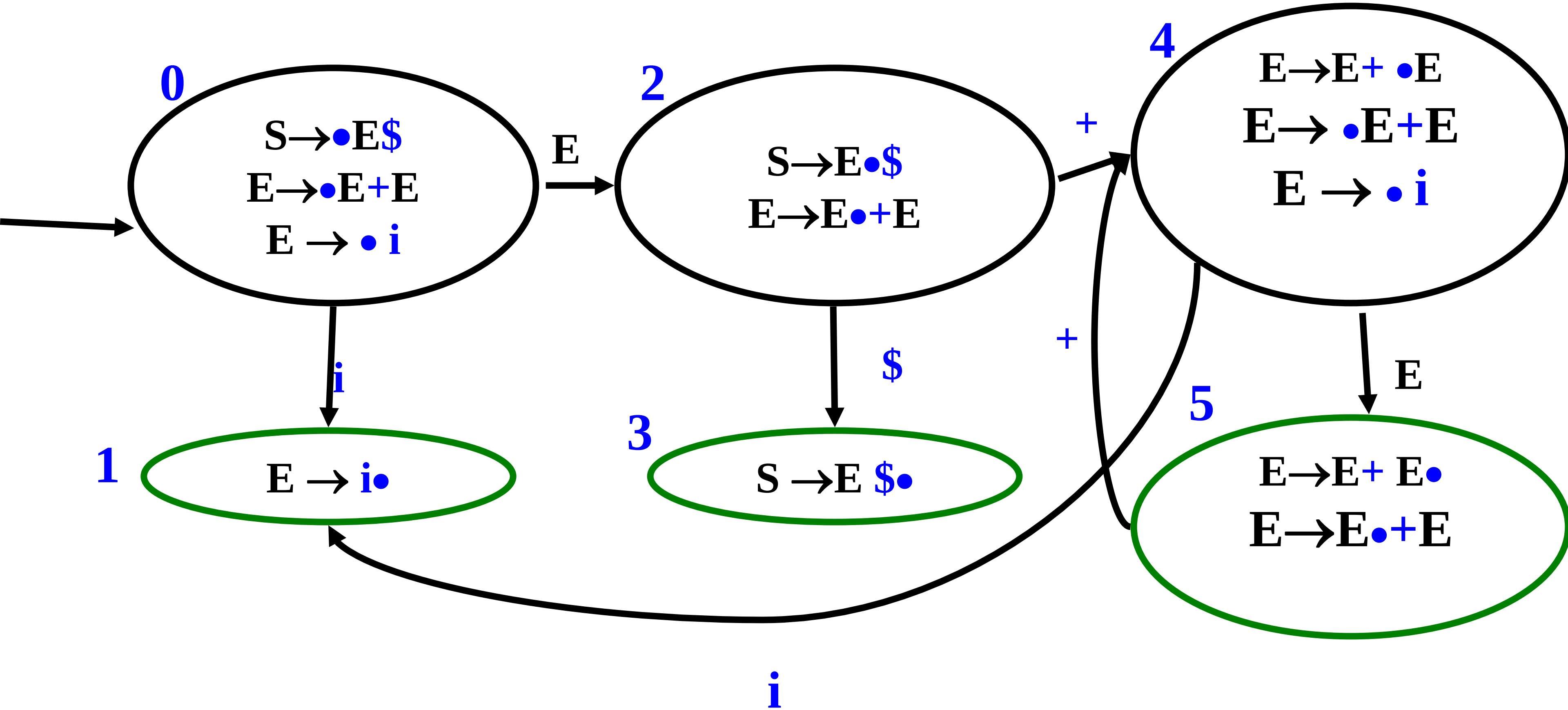
E → E+E

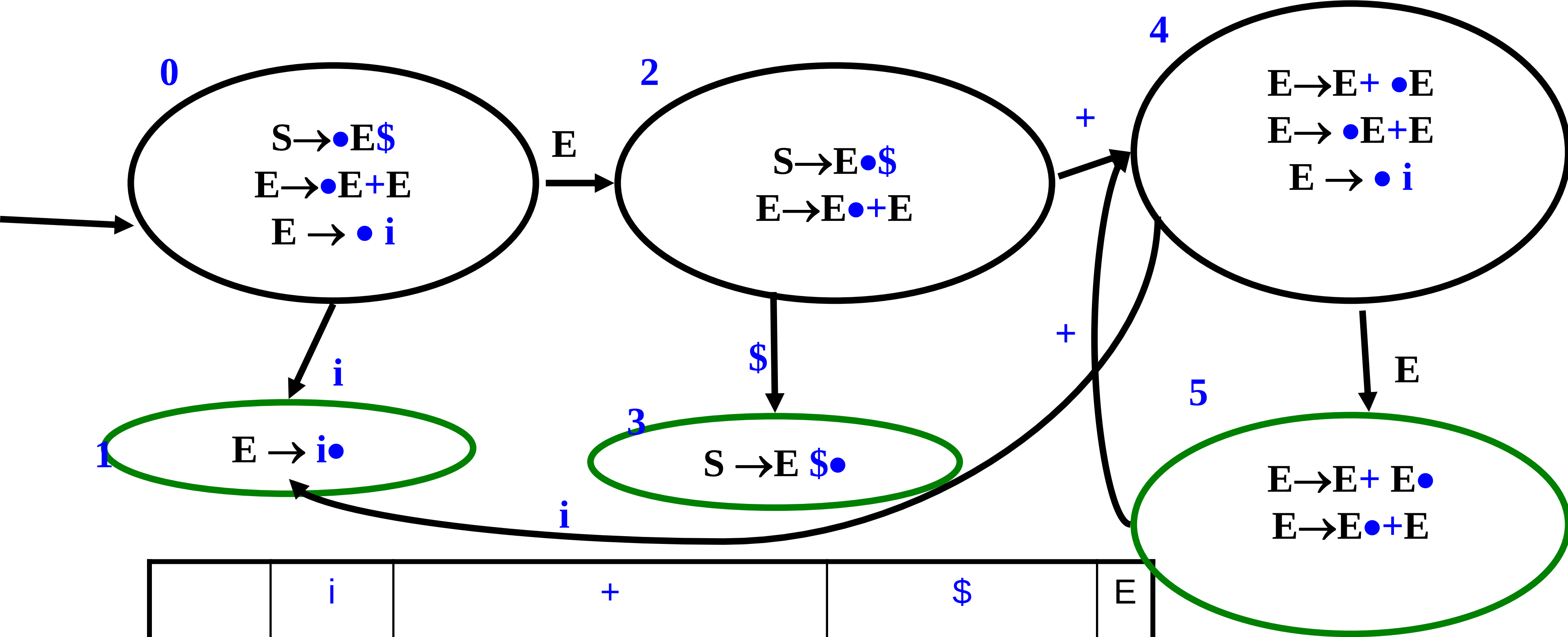
E → i

LR(0) items	i	+	$	E	ε
1: S → •E$				2	4, 8
2: S → E • $			s3		
3: S → E $ •					r **S → E$**
4: E → • E + E				5	4, 8
5: E → E • + E		s6			
6: E → E + • E				7	
7: E → E + E •					r **E → E+E**
8:E → • i	s9				
9:E → i •					r **E → i**

Example Non LR(0) DFA

S → E $ E → E + E | i

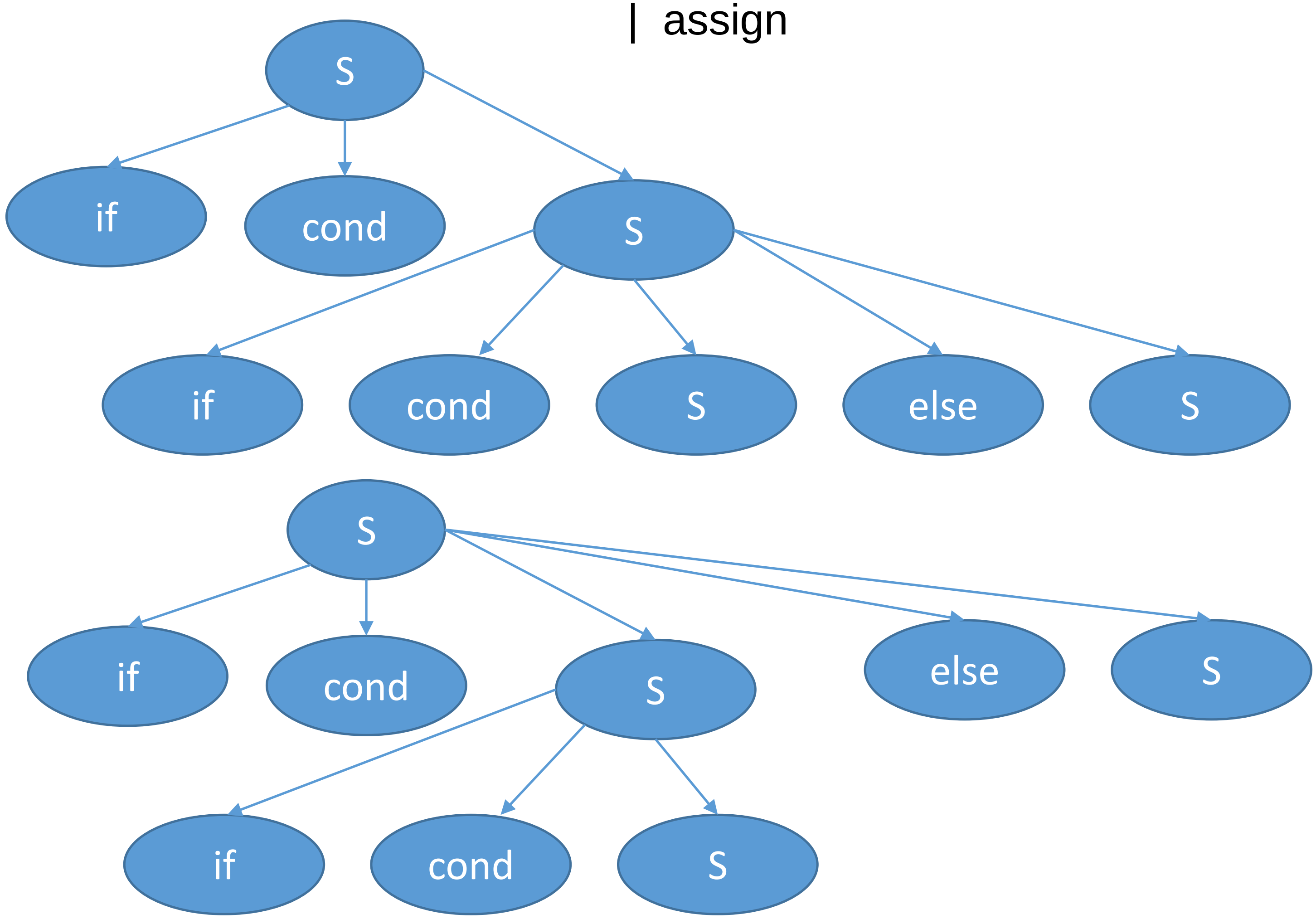




	i	+	$	E
0	s1	err	err	2
1	red E → i			
2	err	s4	s3	
3	accept			
4	s1			5
5	red E → E +E	s4 red E → E +E	red E → E +E	

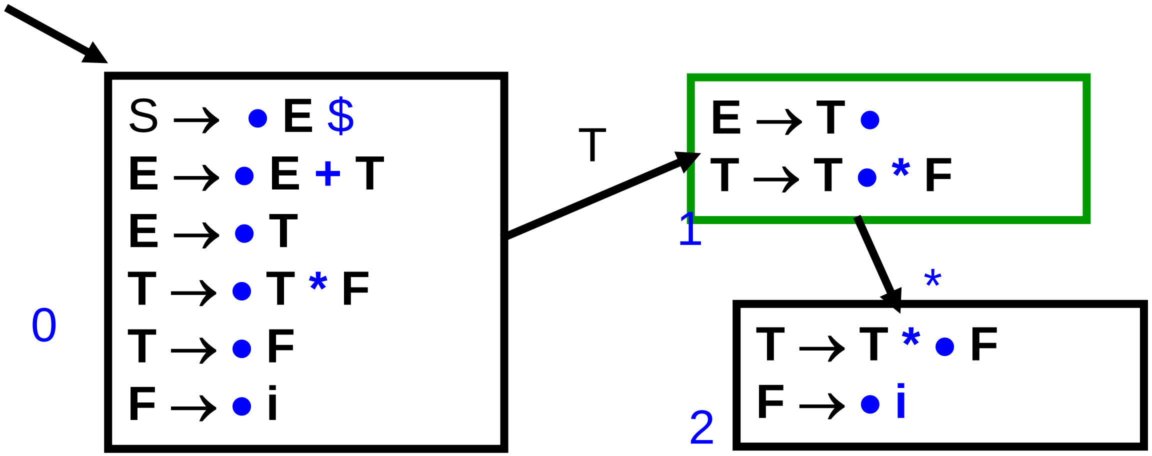
Dangling Else

S → if cond s else s
 | if cond s
 | assign



Non-Ambiguous Non LR(0) Grammar

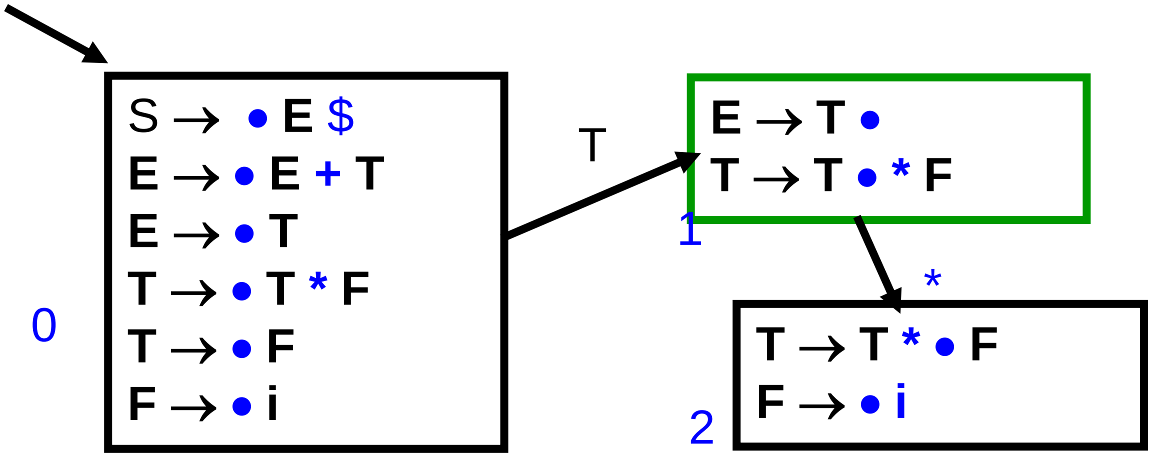
S → E $
E → E + T | T
T → T * F | F
F → i



	i	+	*	
0				
1		?	?	
2				

Non-Ambiguous SLR(1) Grammar

S → **E** $
E → **E** + **T** | **T**
T → **T** * **F** | **F**
F → **i**

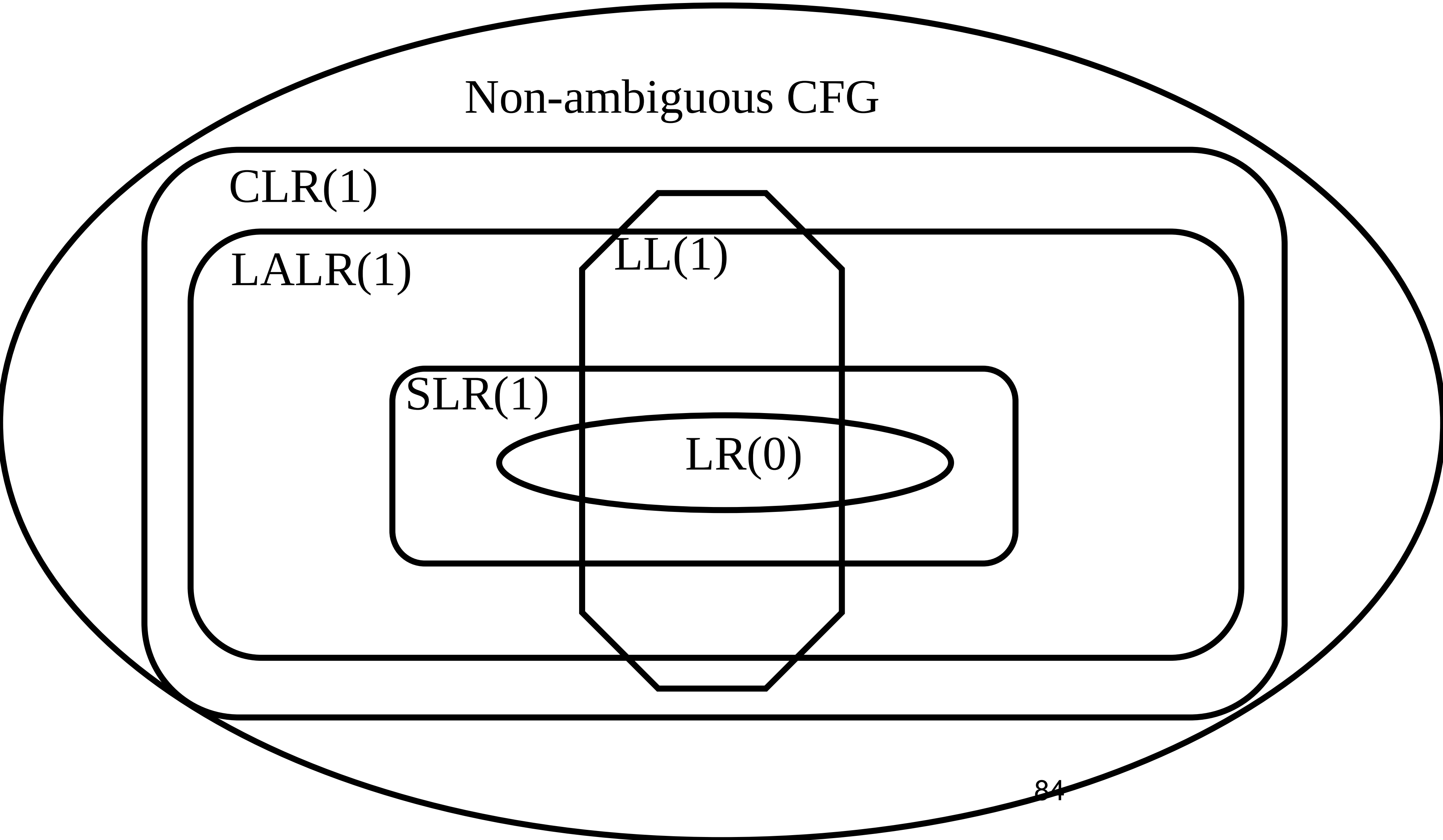


	i	+	*	
0				
1		r **E** → **T**	s2	
2				

LR(1) Parser

- LR(1) Items $A \rightarrow \alpha \bullet \beta$, t
  - α is at the top of the stack and we are expecting βt
- LR(1) State
  - Sets of items
- LALR(1) State
  - Merge items with the same look-ahead

Grammar Hierarchy



Interesting Non LR(1) Grammars

- Ambiguous
  - Arithmetic expressions
  - Dangling-else
- Common derived prefix
  - $A \rightarrow B_1 \; a \; b \mid B_2 \; a \; c$
  - $B_1 \rightarrow \varepsilon$
  - $B_2 \rightarrow \varepsilon$
- Optional non-terminals
  - St $\rightarrow$ OptLab Ass
  - OptLab $\rightarrow$ id : | ε
  - Ass $\rightarrow$ id := Exp

St $\rightarrow$ id: Ass | Ass

Interesting Non LR(1) Grammars

- Ambiguous
  - Arithmetic expressions
  - Dangling-else
- Common derived prefix
  - $A \rightarrow B_1$ a b | B_2 a c
  - $B_1 \rightarrow \varepsilon$
  - $B_2 \rightarrow \varepsilon$
- Optional non-terminals
  - St $\rightarrow$ OptLab Ass
  - OptLab $\rightarrow$ id : | ε
  - Ass $\rightarrow$ id := Exp

A motivating example

- Create a desk calculator
- Challenges
  - Non trivial syntax
  - Recursive expressions (semantics)
    - Operator precedence

Solution (lexical analysis)

```
import java_cup.runtime.*;
%%
%cup
%eofval{
   return sym.EOF;
%eofval}
NUMBER=[0-9]+
%%
”+” { return new Symbol(sym.PLUS); }
”-” { return new Symbol(sym.MINUS); }
”*” { return new Symbol(sym.MULT); }
”/” { return new Symbol(sym.DIV); }
”(” { return new Symbol(sym.LPAREN); }
”)” { return new Symbol(sym.RPAREN); }
{NUMBER} {
        return new Symbol(sym.NUMBER, new Integer(yytext()));
}
\n { }
. { }
```

- Parser gets terminals from the Lexer

```
terminal Integer NUMBER;
terminal PLUS,MINUS,MULT,DIV;
terminal LPAREN, RPAREN;
terminal UMINUS;
nonterminal Integer expr;
precedence left PLUS, MINUS;
precedence left DIV, MULT;
Precedence left UMINUS;
%%
expr ::= expr:e1 PLUS expr:e2
          {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
          | expr:e1 MINUS expr:e2
          {: RESULT = new Integer(e1.intValue() - e2.intValue()); :}
          | expr:e1 MULT expr:e2
          {: RESULT = new Integer(e1.intValue() * e2.intValue()); :}
          | expr:e1 DIV expr:e2
          {: RESULT = new Integer(e1.intValue() / e2.intValue()); :}
          | MINUS expr:e1 %prec UMINUS
          {: RESULT = new Integer(0 - e1.intValue(); :}
          | LPAREN expr:e1 RPAREN
          {: RESULT = e1; :}
          | NUMBER:n
          {: RESULT = n; :}
```

Summary

- LR is a powerful technique
- Generates efficient parsers
- Generation tools exit LALR(1)
  - Bison, yacc, CUP
- But some grammars need to be tuned
  - Shift/Reduce conflicts
  - Reduce/Reduce conflicts
  - Efficiency of the generated parser
- There exist more general methods
  - GLR
  - Arbitrary grammars in n^3
    - Early parsers
    - CYK algorithms