Bottom-Up
Syntax Analysis

Mooly Sagiv

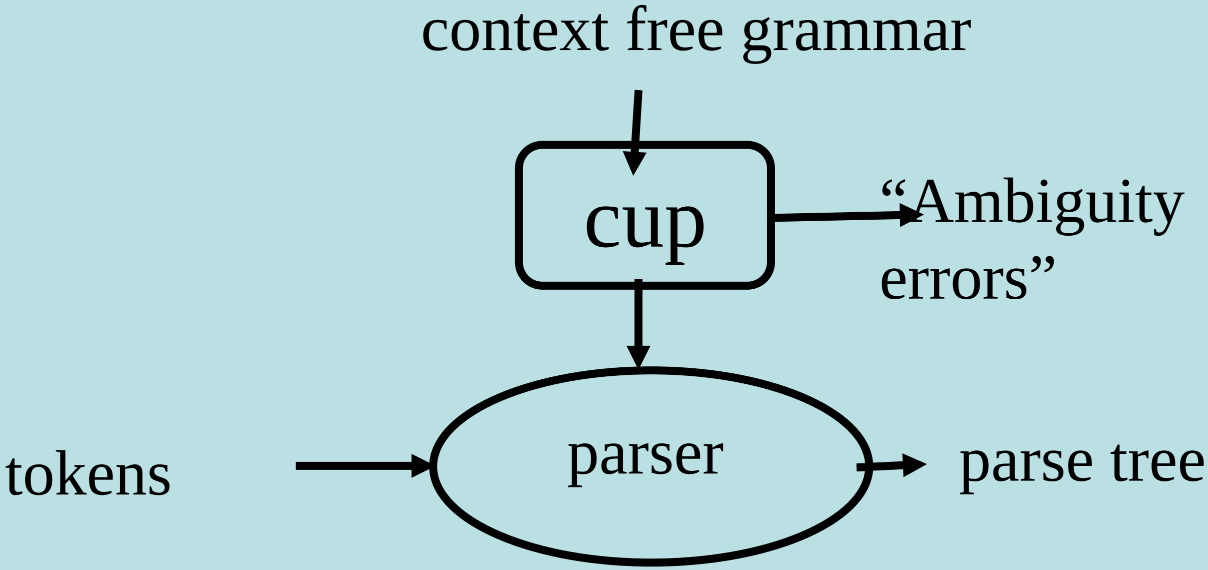
http://www.cs.tau.ac.il/~msagiv/courses/wcc08.html

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

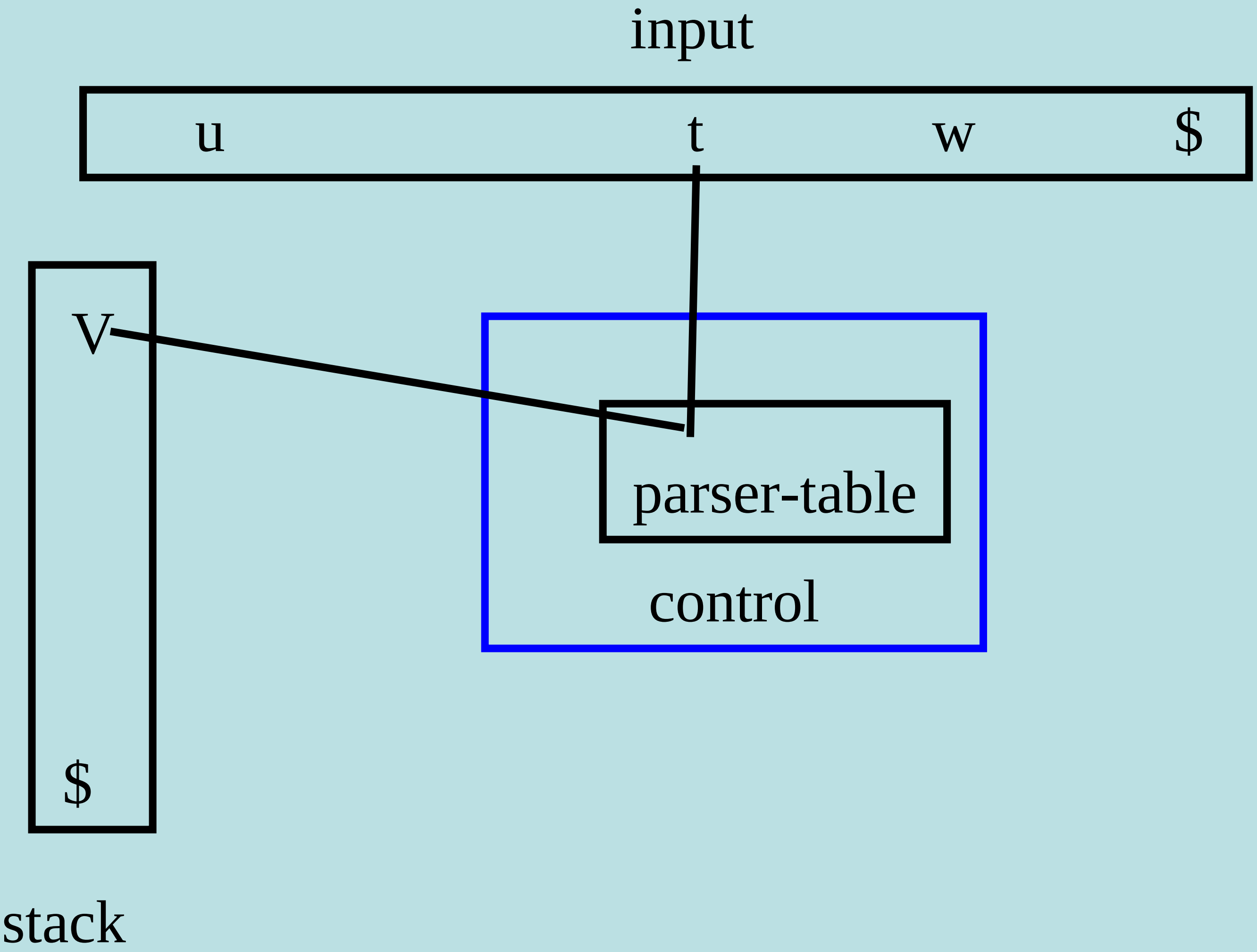
Bottom-Up Syntax Analysis

- 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

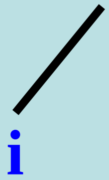


Informal Example(1)

S → E$ E → T | E + T   T → i | (E)

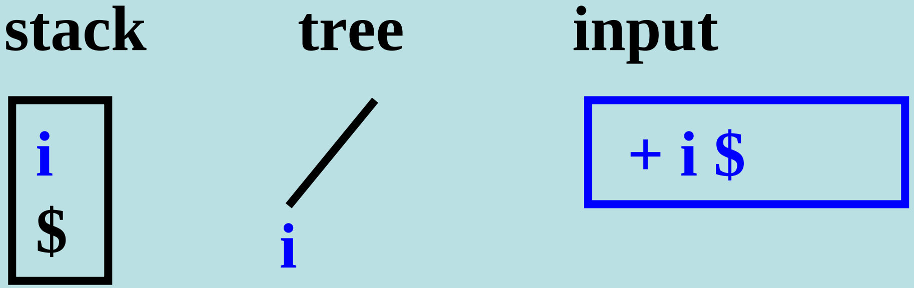
stack	tree	input
$		i + i $

shift

stack	tree	input
i $		+ i $

Informal Example(2)

S → E$ E → T | E + T   T → i | (E)



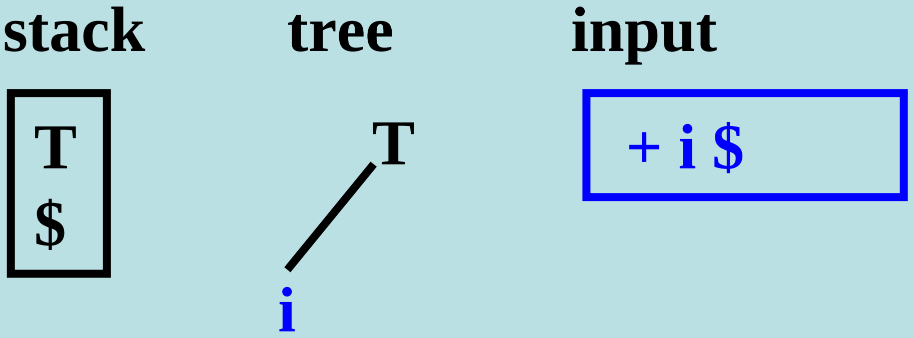
stack   tree   input

i
$

i

+ i $

reduce T → i



stack   tree   input

T
$

T
i

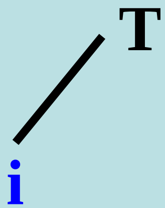
+ i $

Informal Example(3)

S → E$ E → T | E + T   T → i | (E)

stack **tree** **input**

T
$

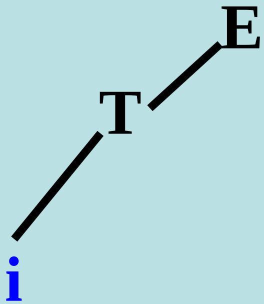


+ i $

reduce E → T

stack **tree** **input**

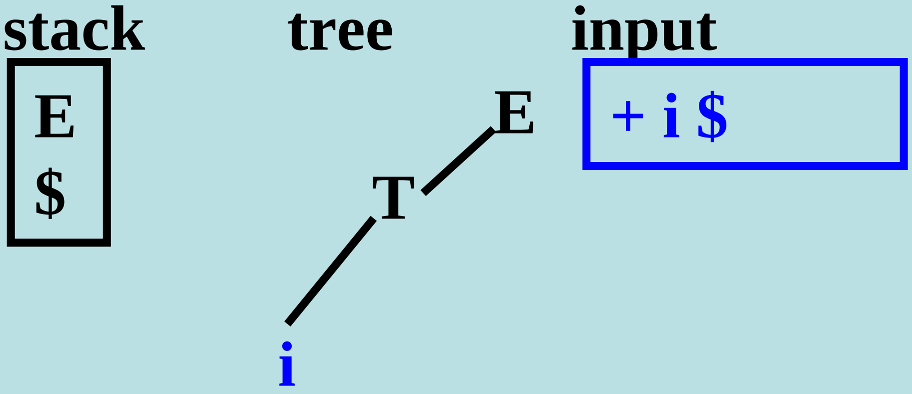
E
$



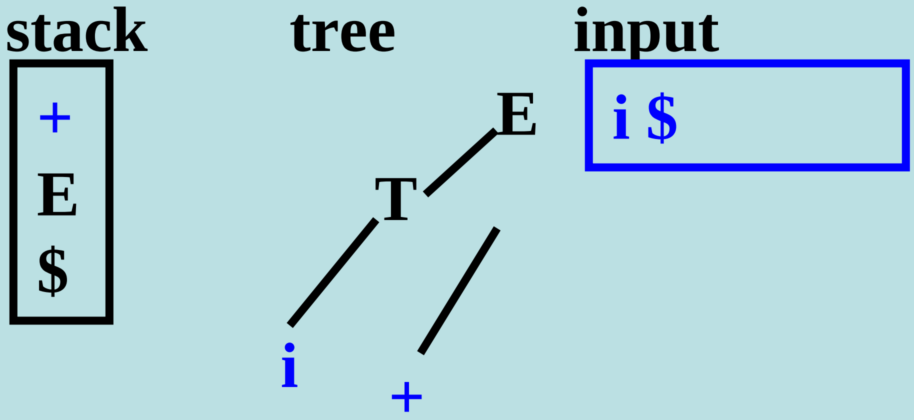
+ i $

Informal Example(4)

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



shift

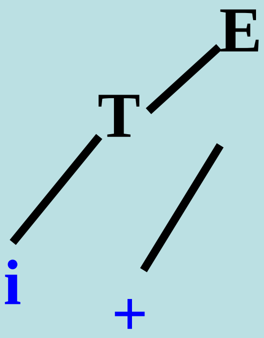


Informal Example(5)

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

stack tree input

+
E
$

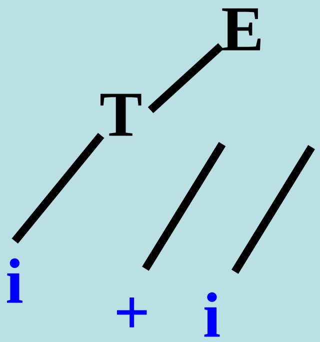


i $

shift

stack tree input

i
+
E
$

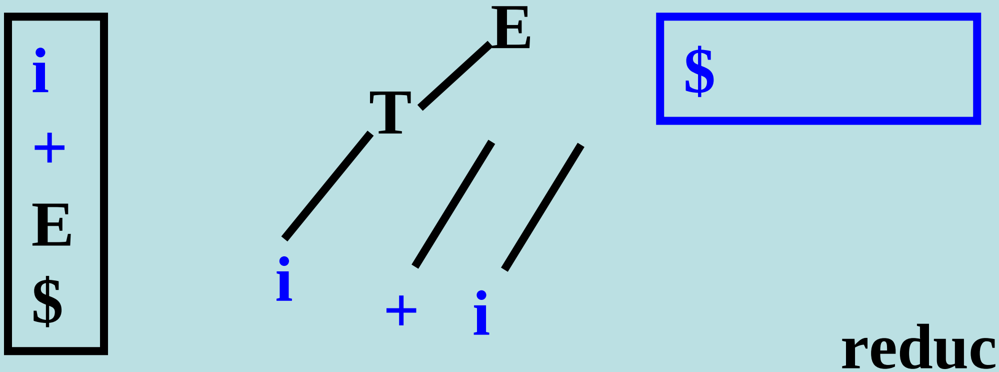


$

Informal Example(6)

S → E$ E → T | E + T   T → i | (E)

stack   tree   input



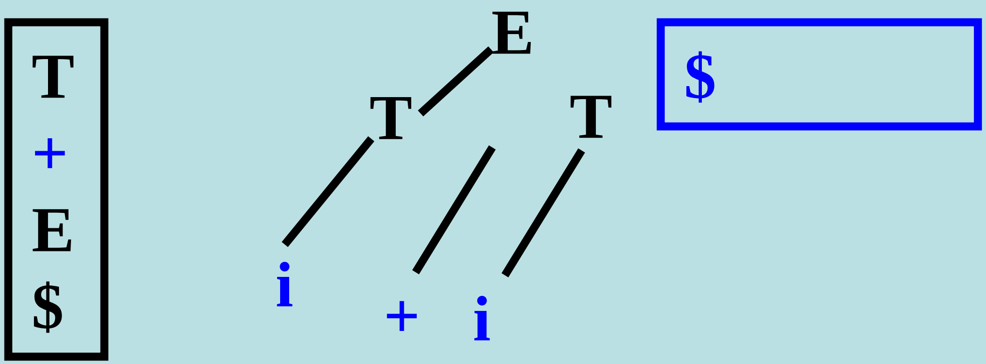
i
+
E
$

E
T
i
+
i

$

reduce T → i

stack   tree   input



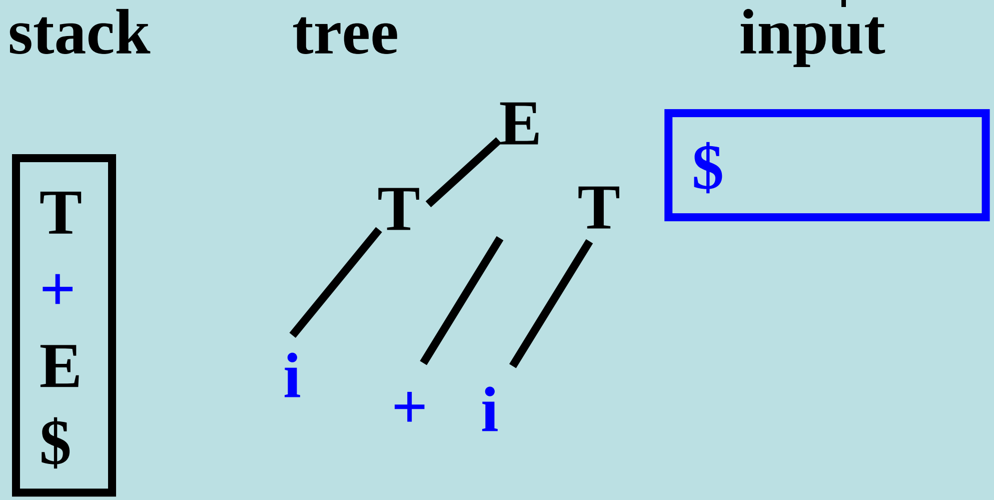
T
+
E
$

E
T
T
i
+
i

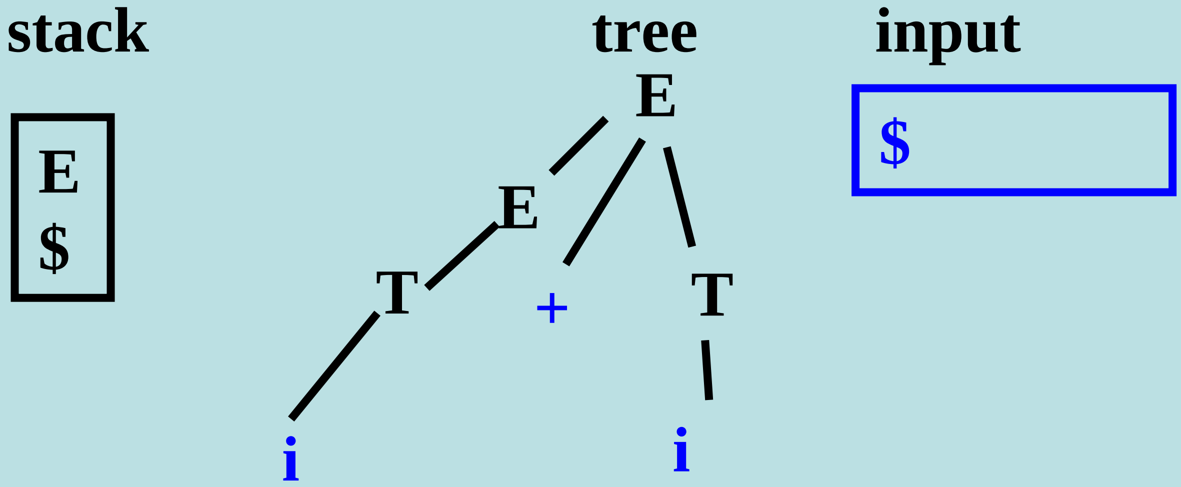
$

Informal Example(7)

S → E$ E → T | E + T   T → i | (E)



reduce E → E + T



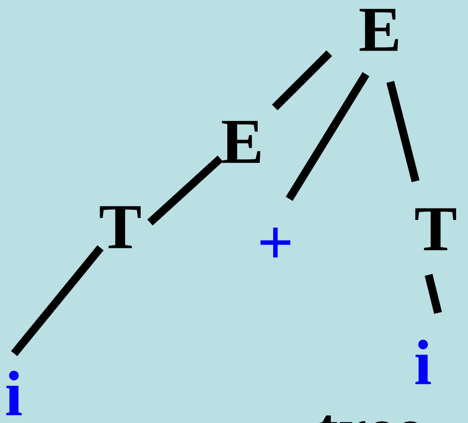
Informal Example(8)

S → E$ E → T | E + T T → i | (E)

stack

E
$

tree



input

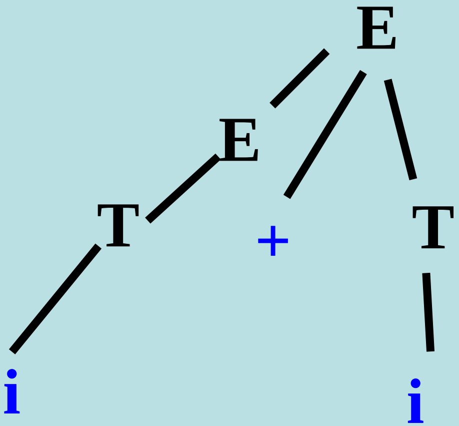
$

shift

stack

$
E
$

tree

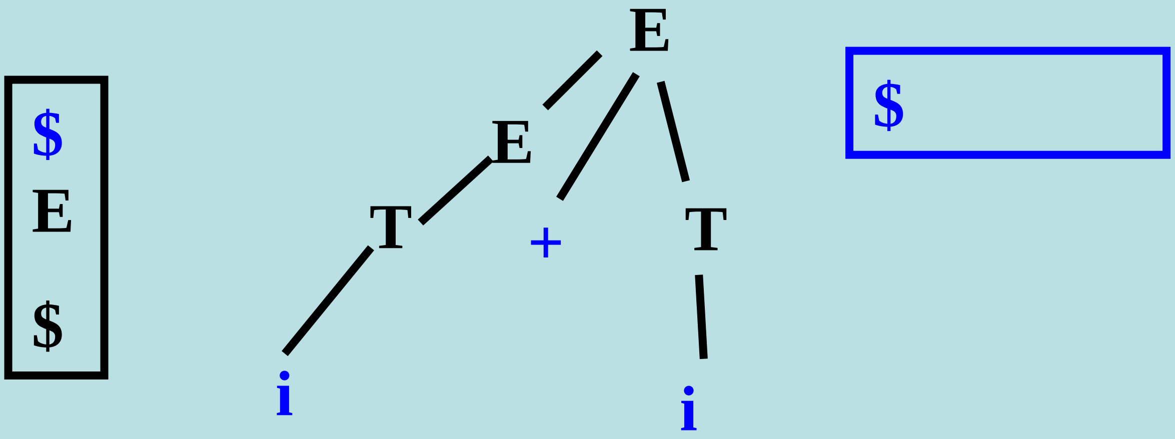


input

Informal Example(9)

S → E$ E → T | E + T   T → i | (E)

stack   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

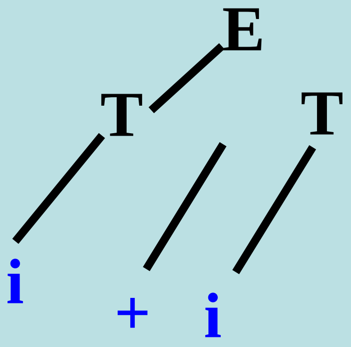
- Deciding between shift and reduce

Informal Example(7)

S → E$ E → T | E + T   T → i | (E)

stack   tree   input

T
+
E
$

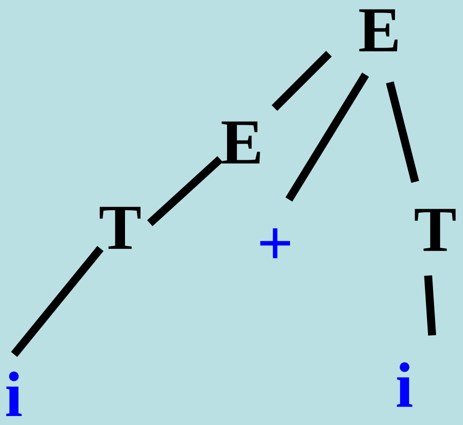


$

reduce E → E + T

stack   tree   input

E
$

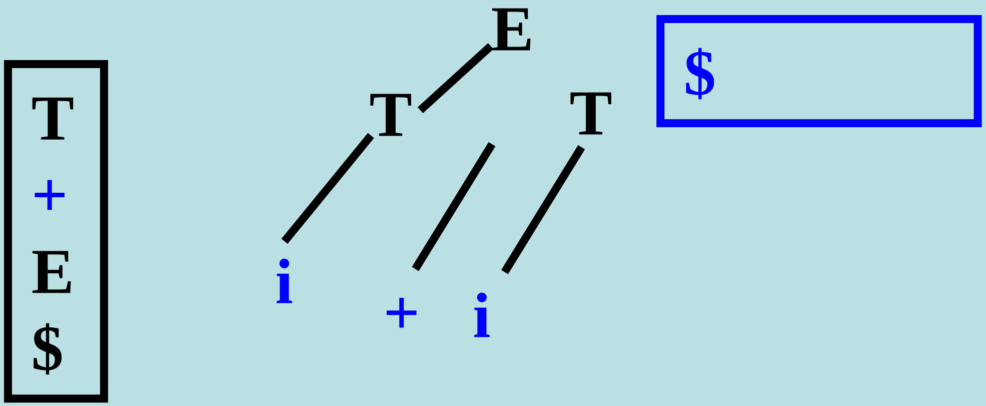


$

Informal Example(7')

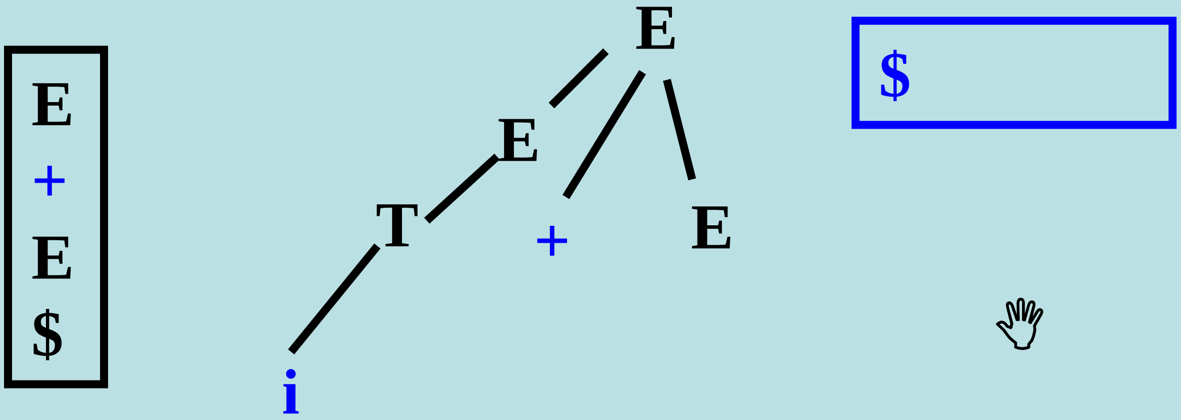
S → E$ E → T | E + T T → i | (E)

stack tree input

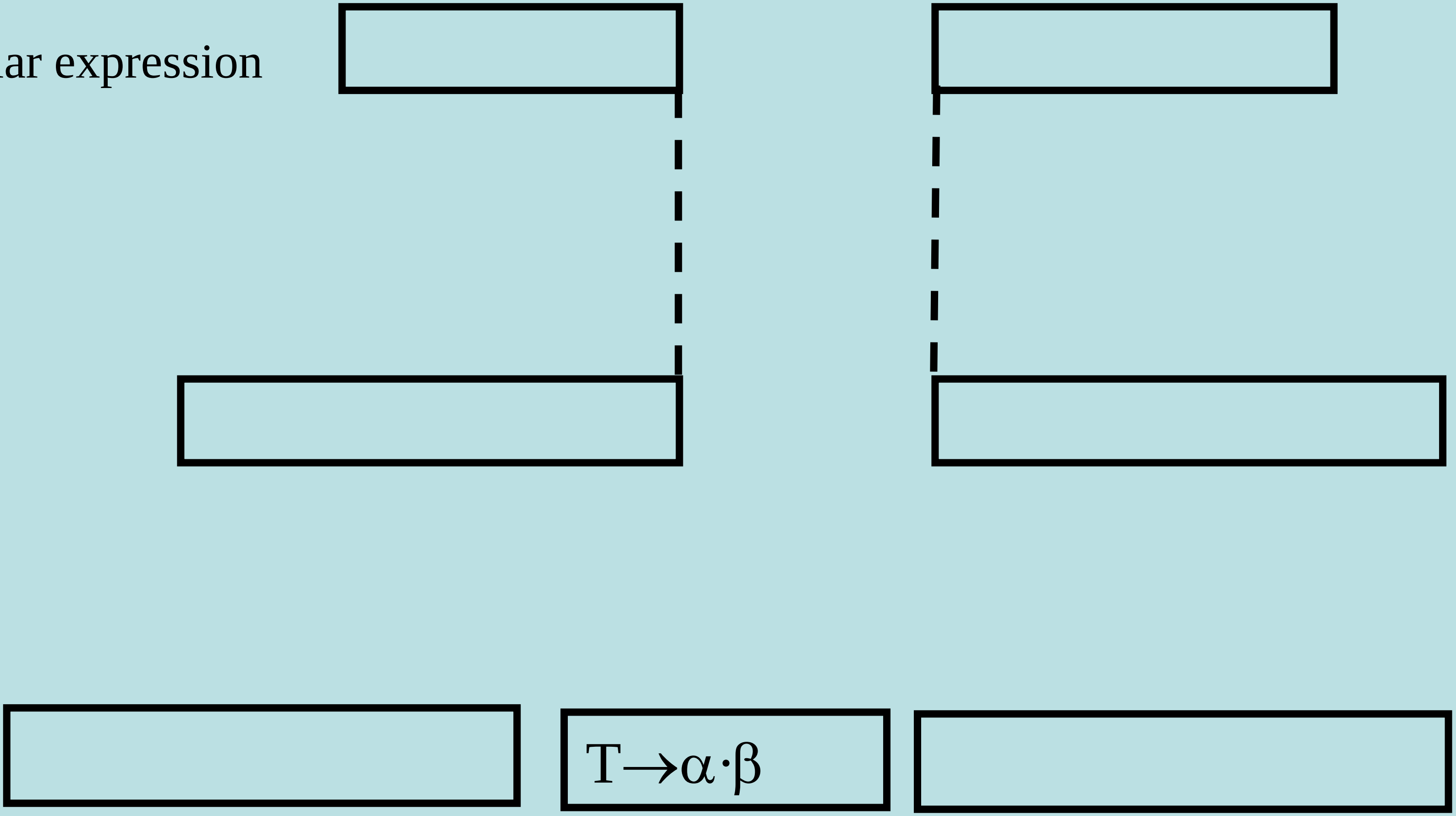


reduce E → T

stack tree input



Bottom-UP LR(0) Items



already matched

still need to be matched

regular expression

input

input

$T \rightarrow \alpha \cdot \beta$

already matched α

expecting to match β

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 → E$ E → T | E + T   T → i | (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)

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

stack

$11: T \to i \bullet$

$10: T \to \bullet i$

$4: E \to \bullet T$

$6: E \to \bullet E+T$

$1: S \to \bullet E\$$

input

$+ \; i \; \$$

reduce $T \to i$

stack

$5: E \to T \bullet$

$4: E \to \bullet T$

$6: E \to \bullet E+T$

$1: S \to \bullet E\$$

input

$+ \; i \; \$$

Formal Example(6)

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

stack **input**

5: $E \rightarrow T \bullet$

4: $E \rightarrow \bullet T$

6: $E \rightarrow \bullet E+T$

1: $S \rightarrow \bullet E\$$

+ i $

reduce $E \rightarrow T$

stack **input**

7: $E \rightarrow E \bullet +T$

6: $E \rightarrow \bullet E+T$

1: $S \rightarrow \bullet E\$$

+ 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 → E$ E → T | E + T   T → i | (E)

stack   input

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

i $

stack   input   ε-move 10

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

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 → E$ E → T | E + T   T → i | (E)

stack

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

input

$

reduce T → i

stack

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

input

$

Formal Example(11)

S → E$ E → T | E + T   T → i | (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)

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

stack

2: $S \to E \bullet \$$

1: $S \to \bullet E\$$

input

$

shift 3

stack

3: $S \to E \$ \bullet$

2: $S \to E \bullet \$$

1: $S \to \bullet E\$$

input

Formal Example(13)

S → E$ E → T | E + T T → i | (E)

stack **input**

3: S → E $ •

2: S → E • $

1: S → •E$

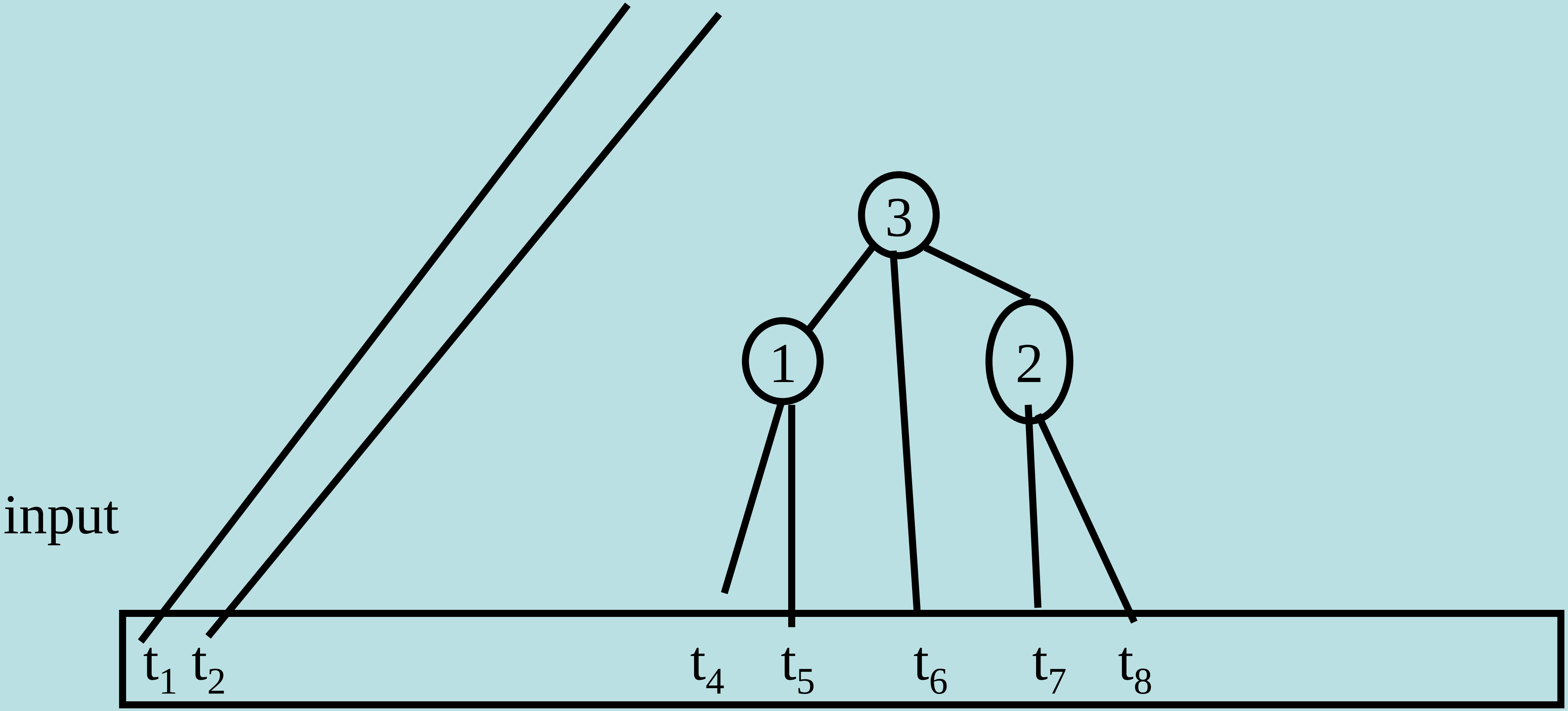
reduce S → E$

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

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

S → E$

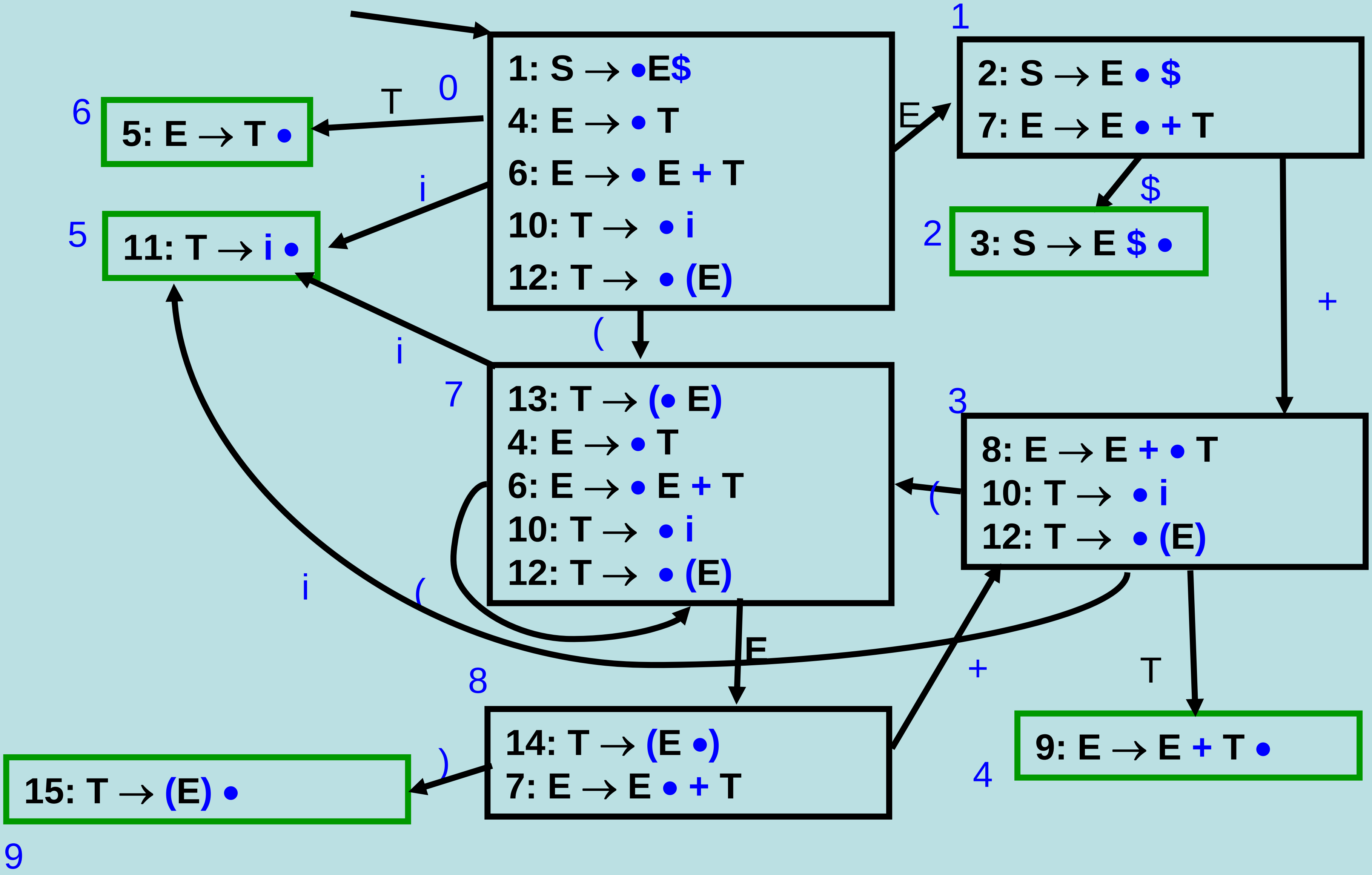
E → T

E → E + T

T → i

T →(E)

	(	)	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



1: S → •E$
4: E → • T
6: E → • E + T
10: T → • i
12: T → • (E)

2: S → E • $
7: E → E • + T

5: E → T •

11: T → i •

3: S → E $ •

13: T → (• E)
4: E → • T
6: E → • E + T
10: T → • i
12: T → • (E)

8: E → E + • T
10: T → • i
12: T → • (E)

14: T → (E •)
7: E → E • + T

15: T → (E) •

9: E → E + T •

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)						

	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

1 (E)

0 ($)

input

$

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

	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)

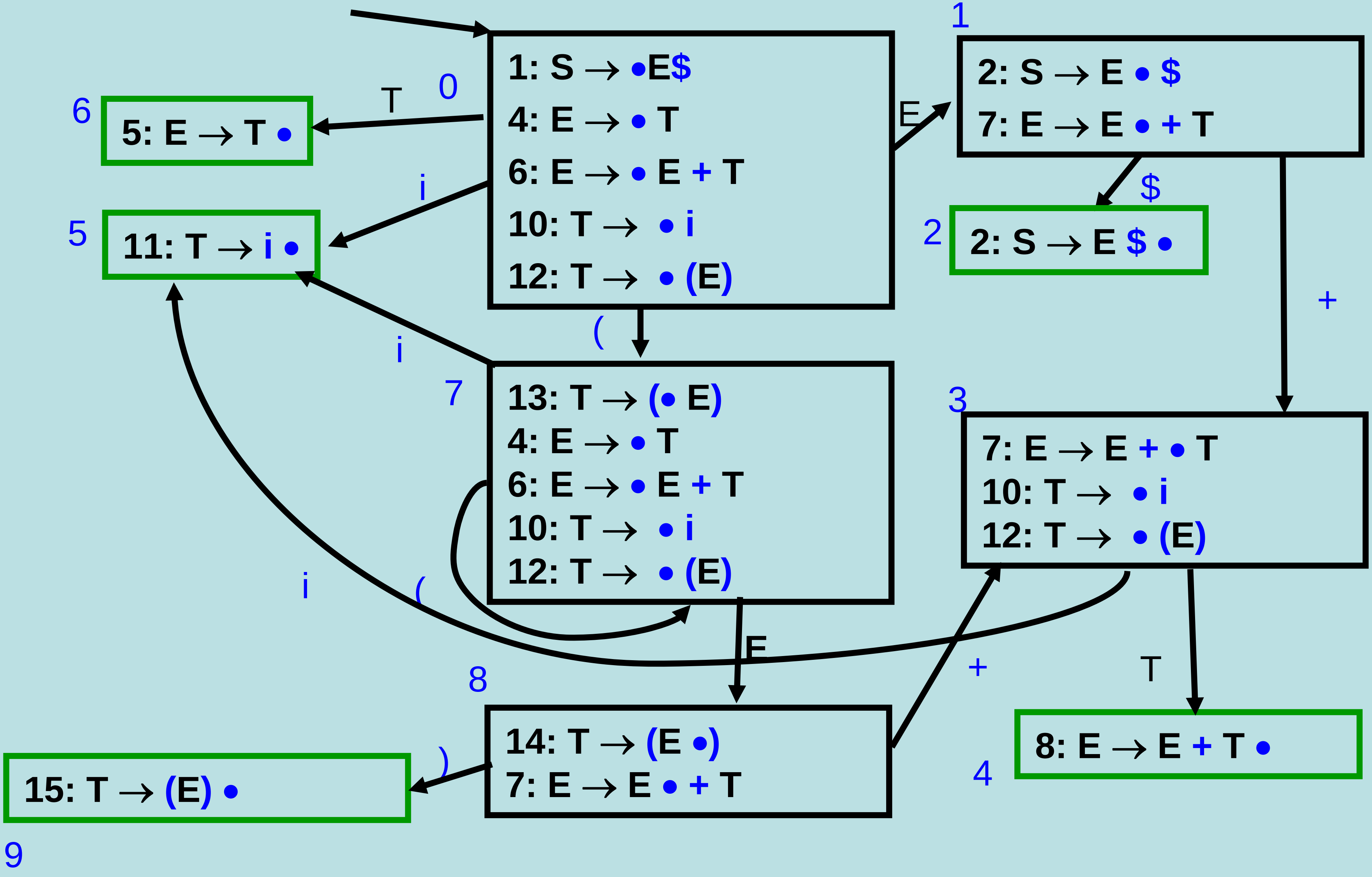
7(()

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

2

2: S → E $ •

6

5: E → T •

5

11: T → i •

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)

4

8: E → E + T •

8

14: T → (E •)

7: E → E • + T

9

15: T → (E) •

Constructing LR(0) parsing table

- Add a production S' → S$
- Construct a 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
  - $A \rightarrow \alpha \bullet t \beta \in s_i$
- $\text{Goto}(s_i, X) = s_j$
  - $A \rightarrow \alpha \bullet X \beta \in s_i$
  - $\delta(s_i, X) = s_j$
- 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$

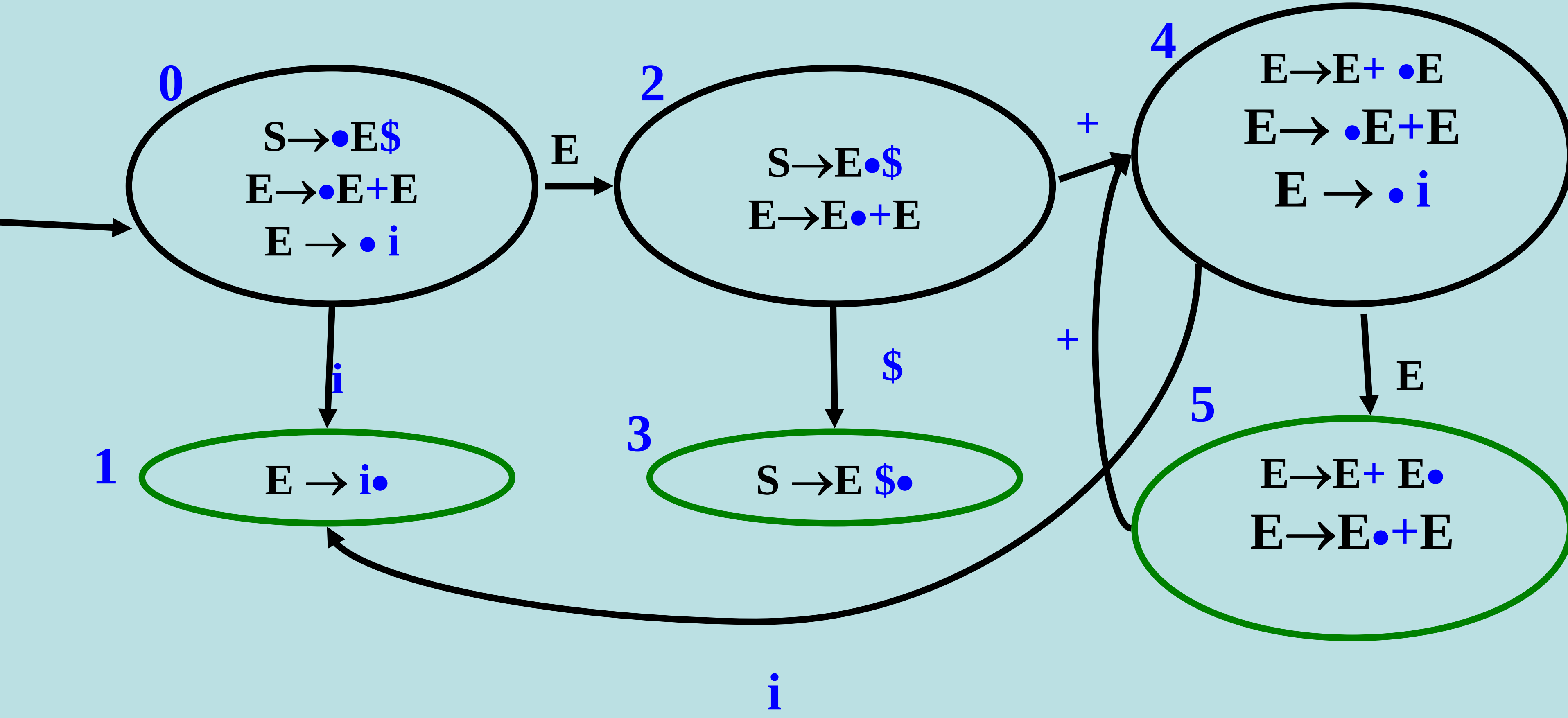
E → E+E

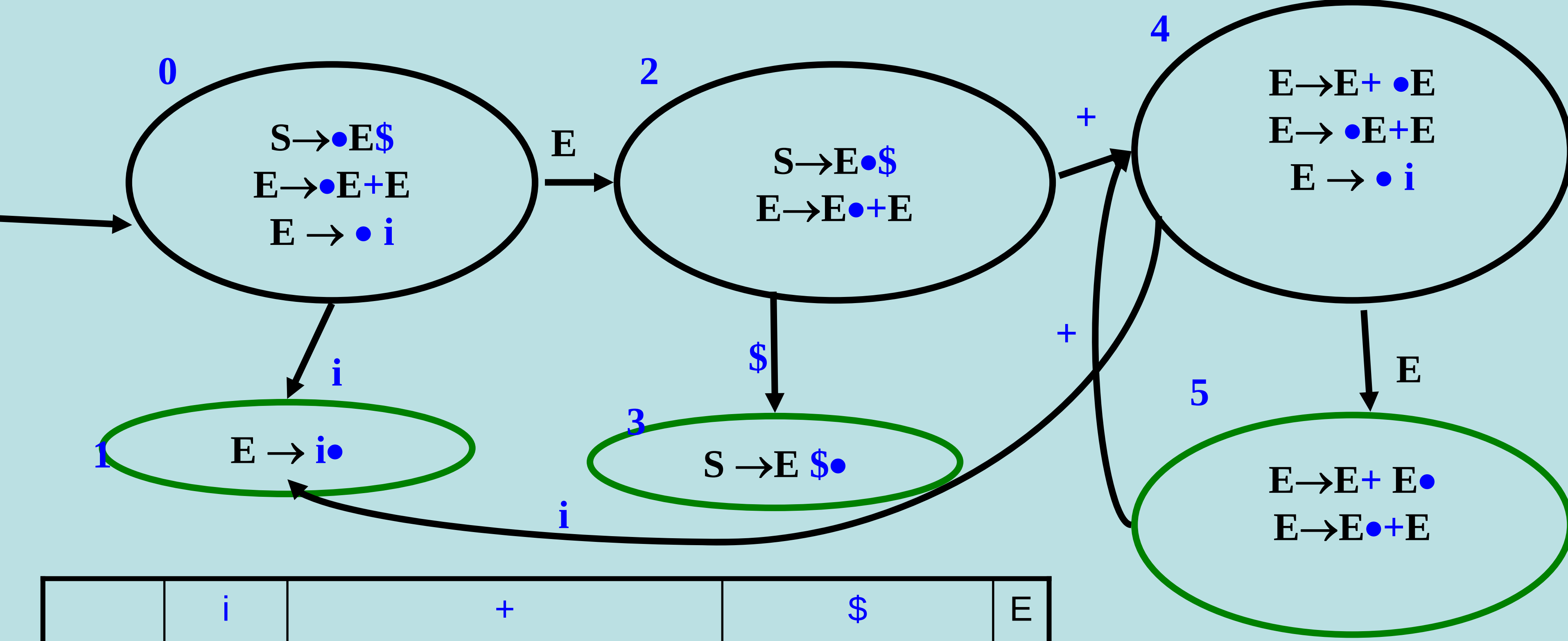
E → i

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

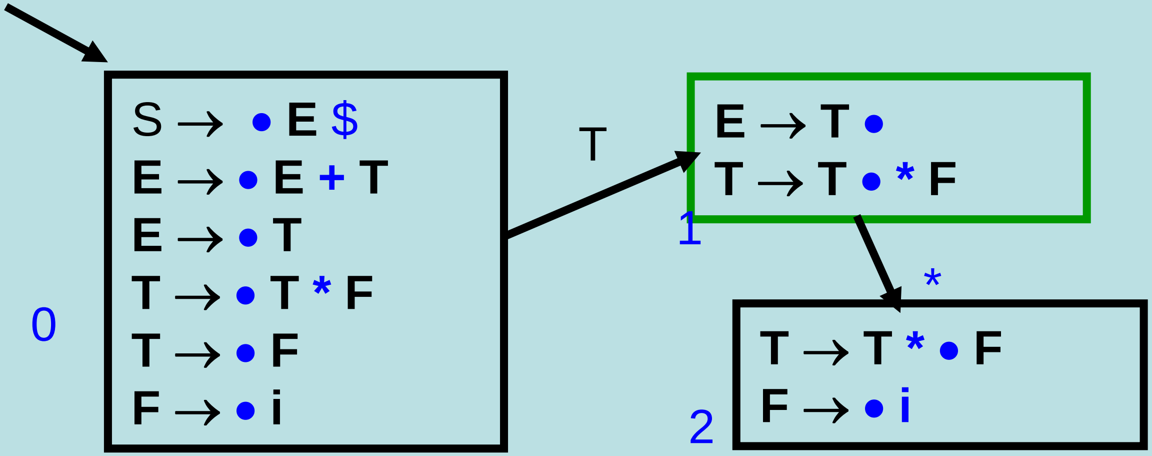




	i	+	$	E
0	s1	err	err	2
1	red E → i			
2	err	s4	s3	
3	accept			
4	s1			5
5	red E → E +E	s4 red E → E +E	red E → E +E	

Non-Ambiguous Non LR(0) Grammar

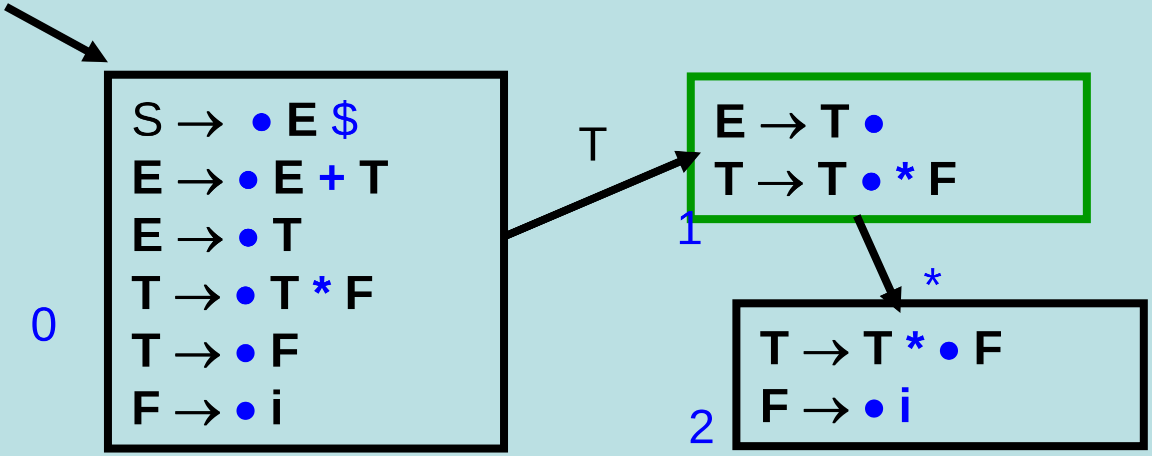
S → E $
E → E + T | T
T → T * F | F
F → i



	i	+	*	
0				
1		?	?	
2				

Non-Ambiguous SLR(1) Grammar

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

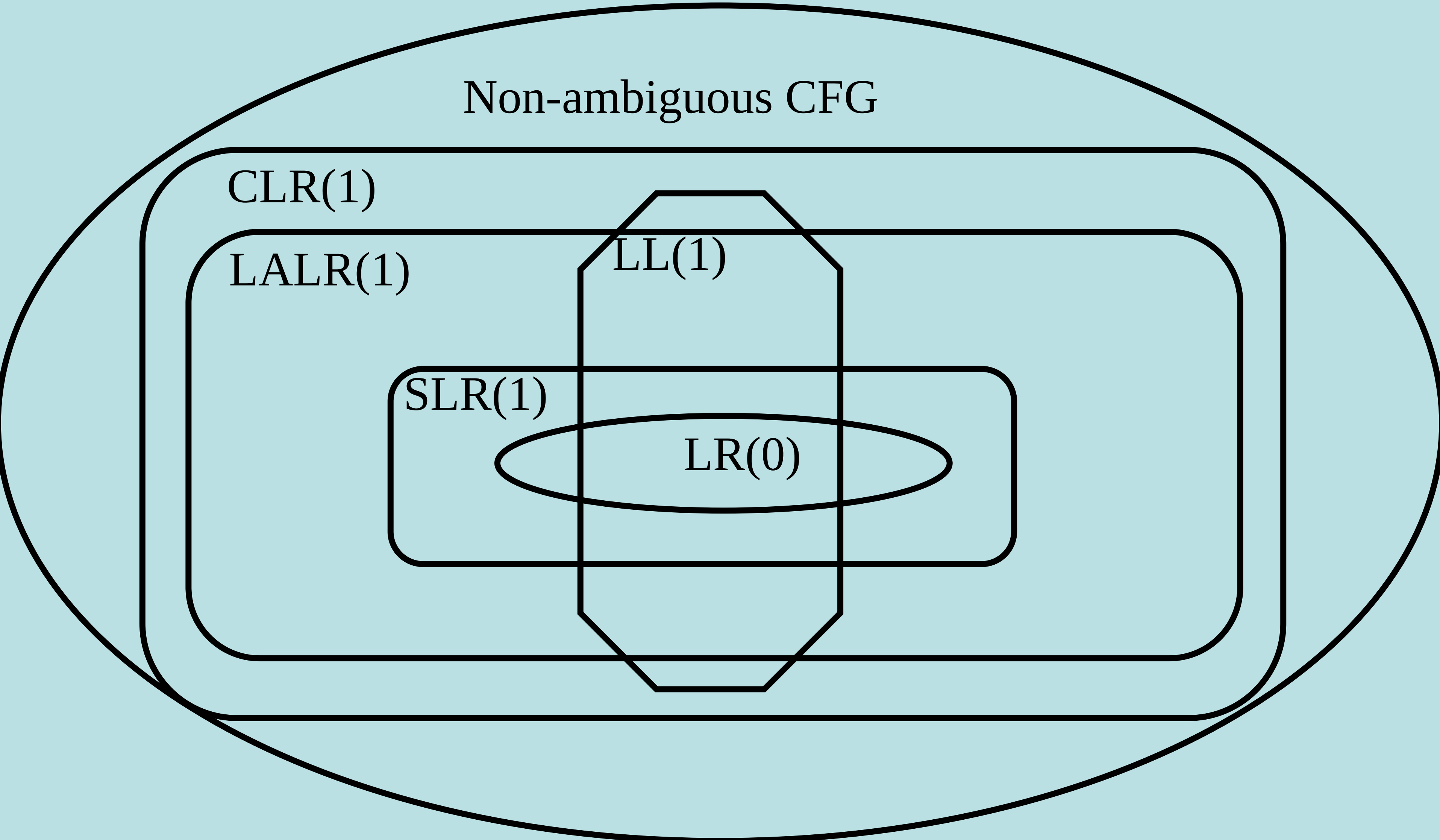


	i	+	*	
0				
1		r E → T	s2	
2				

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 → OptLab Ass
  - OptLab → id : | ε
  - Ass → 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