Compiler Construction

ARNOLD MEIJSTER
A.MEIJSTER@RUG.NL

LL Grammar Restriction 1

- **Grammar Restriction 1**:

An LL grammar contains no left-recursive rules.

LL Grammar Restriction 2

- **Grammar Restriction 2**:

The First sets of all alternatives/choices for the same LHS must be different (so we know which path to take upon seeing the next terminal symbol/token).

LL Grammar Restriction 3

- **Grammar Restriction 3:**

If a nonterminal may occur zero times (i.e. is optional), its First and Follow sets must be different (so we know whether to parse it or skip it on seeing a terminal symbol/token).

Recursive Descent Parsing

- Massage grammar to meet the LL(1) conditions
  - Remove left recursion
  - Left factor, where possible

- Define a procedure for each non-terminal
  - Implement a case for each right-hand side
  - Call procedures for non-terminals
  - Add extra code, as needed

Table-driven approach LL(1)

- Encode grammar in a table
  - Row for each non-terminal
  - Column for each terminal symbol

  Table[NT, symbol] = rule

expr2 → **+ *term expr2***
| *ε*
term2 → *** *factor term2***
| *ε*
factor → `number`

	+	*****	**number**
expr2	**+ *term expr2***	**Error (?)**	**Error (?)**
term2	**Error (?)**	*** *factor term2***	**Error (?)**
factor	**error**	**error**	***number***

How to Construct Parse Tables?

- Consider a production X → β
- Add → β to the X row for each symbol in FIRST(β)
- If β can derive ε, add → ε for each symbol in Follow(X)

S → E S'
S' → ε | + S
E → num | (S)

First(S) = {num, (}
First(S') = {ε, + }
First(E) = { num, (}

Follow(S) = { $,) }
Follow(S') = { $,) }
Follow(E) = { +,), $ }

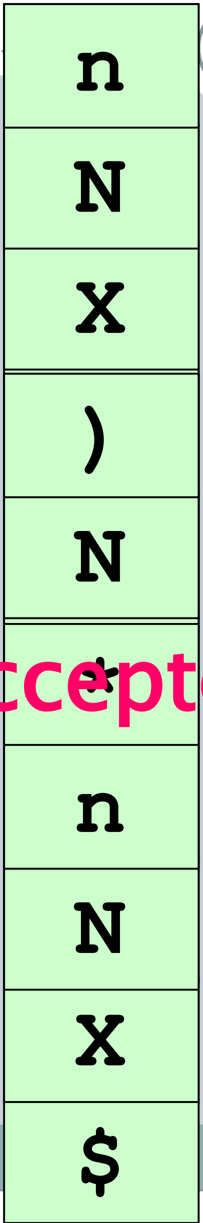
	num	+	(	)	$
S	E S'		E S'		
S'		+ S		ε	ε
E	num		(S)		

Table driven LL(1) parser code

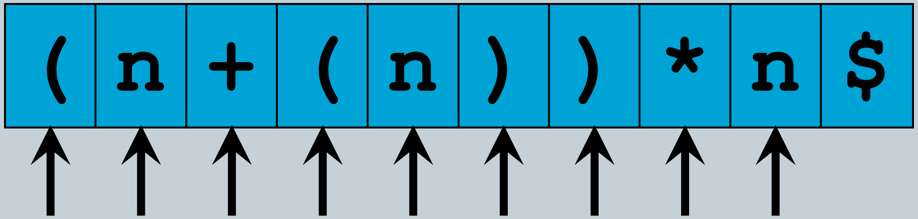
```
push EOF and the start symbol onto Stack
top ← top of Stack
loop forever
   if top = EOF and token = EOF then break & report success
   if top is a terminal then
      if top matches token then
         pop Stack                              /* recognized top */
         token ← next_token()
      else
         syntax error
   else                                         /* top is a non-terminal */
      if TABLE[top,token] is $B_1B_2…B_k$ then
         pop Stack                              /* get rid of non-terminal (lhs) */
         push Bk, Bk-1, …, B1                   /* in that order */
   top ← top of Stack
```

Example of table driven LL(1) Parsing

E
⇒TX
⇒FNX
⇒ (E) NX
⇒ (TX) NX
⇒ (FNX) NX
⇒ (nNX) NX
⇒ (nX) NX
⇒ (nATX) NX
⇒ (n+TX) NX
⇒ (n+FNX) NX
⇒ (n+ (E) NX) NX
⇒ (n+ (TX) NX) NX
⇒ (n+ (FNX) NX) NX
⇒ (n+ (nNX) NX) NX
⇒ (n+ (nX) NX) NX
⇒ (n+ (n) NX) NX
⇒ (n+ (n) X) NX
⇒ (n+ (n)) NX
⇒ (n+ (n)) MFNX
⇒ (n+ (n)) *FNX
⇒ (n+ (n)) *nNX
⇒ (n+ (n)) *nX
⇒ (n+ (n)) *n



Accepted



E → T X
X → A T X | ε
A → + | -
T → F N
N → M F N | ε
M → *
F → (E) | n

Problematic grammars

- Sometimes, it is impossible to convert a grammar into an equivalent grammar that is LL(1).

- Consider the following grammar:

  $S \rightarrow a\ B\ c\ S\ T \mid d$

  $T \rightarrow e\ S \mid \varepsilon$

  $B \rightarrow b$

Consider the input: a b c a b c d e d

Problematic grammars

S → a B c S T | d
T → e S | ε
B → b

We can derive "a b c a b c d e d" as follows:

S → a B c S T
→ a b c S T
→ a b c a B c S T T
→ a b c a b c S T T
→ a b c a b c d T T

The next input token is an 'e'. Now, we have two continuations:

a b c a b c d T T →a b c a b c d e S T →a b c a b c d e d T → a b c a b c d e d
a b c a b c d T T →a b c a b c d T → a b c a b c d e S → a b c a b c d e d

No matter how hard you try, you will not be able to convert this grammar into an unambiguous LL(1) equivalent.

Problematic grammars: parse table

- S → a B c S T | d
- T → e S | ε
- B → b

	a	b	c	d	e
S	**A B c S T**	**error**	**error**	**d**	**error**
T	**error**	***error***	**error**	**error**	**e S** **ε**
B	**error**	**b**	***error***	***error***	***error***

- ‘Solutions’:
  - #1: table[T,e] = e S
  - #2: table[T,e]= ε

- These ‘solutions’ implement a preference.
  - Note that ‘solution’ #2 changes the accepted language (‘e’ is never accepted)

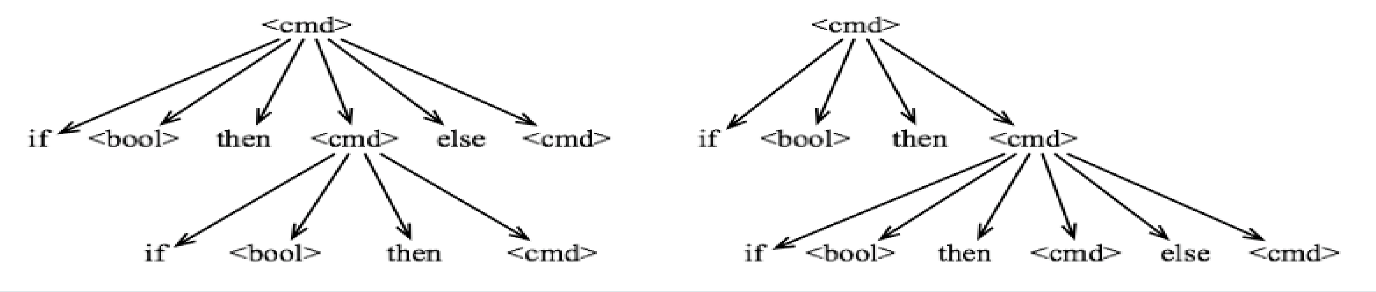
Dangling-else problem

cmd → **if (** bool**)** **then** cmd **else** cmd | **if (**bool**)** cmd

- Grammar can be left-factored (but stays problematic):

  cmd → **if (** bool**) then** cmd elsePart

  elsePart → **else** cmd | ε



Problem: **`if <bool> then if <bool> then <cmd> else <cmd>`**

Standard 'solution':

`table[elsePart,else]=` **else cmd**

which corresponds with the convention to pair the **else** with the closest **if**.

LL(k) solves our problems?

- Sometimes, it is possible to resolve problems using more than one symbol look-ahead.
  - LL(k): k symbols look-ahead

- Consider the grammar:

  1: $S \rightarrow Ab$
  2: $S \rightarrow Bb$
  3: $A \rightarrow aa$
  4: $B \rightarrow aaa$

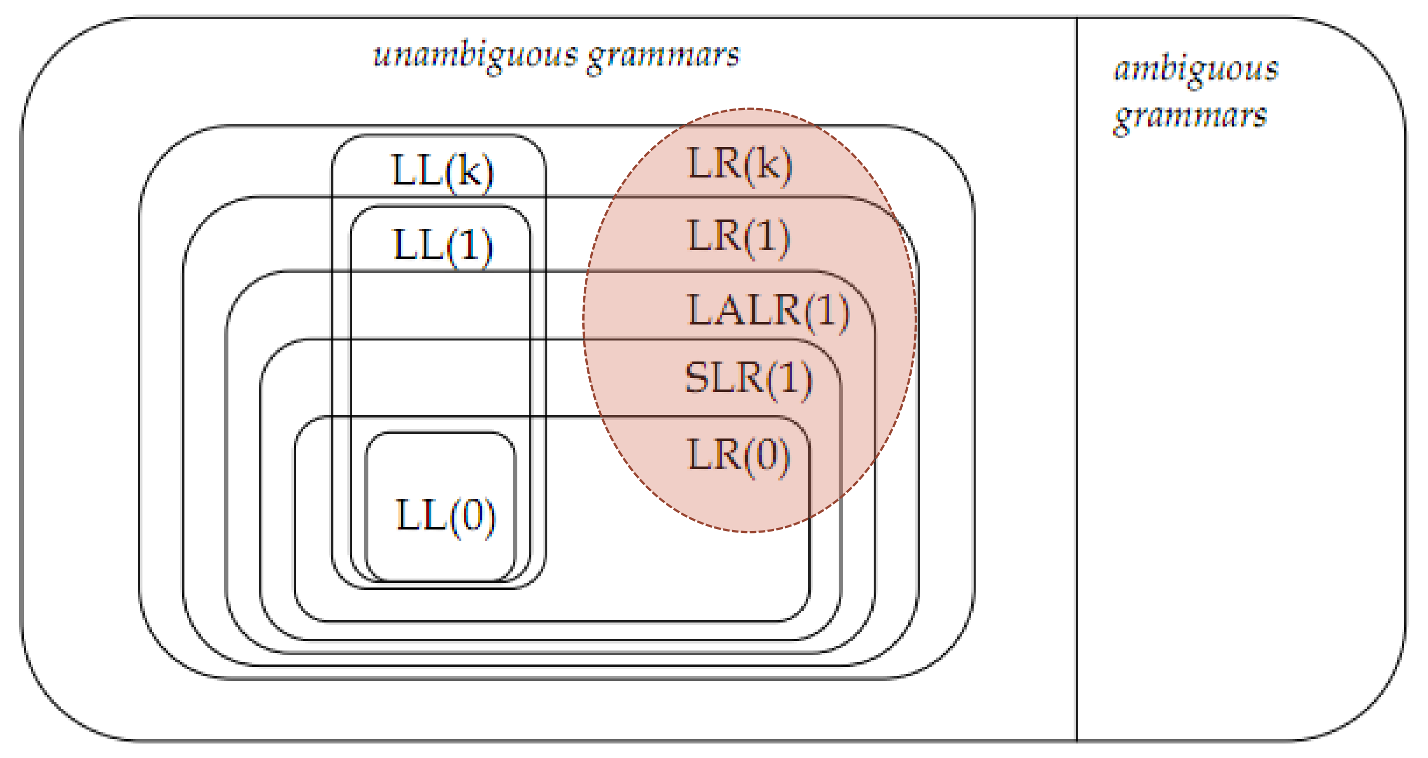
- It can only produce *aab* or *aaab*.

- The grammar is not $LL(1)$, nor is it $LL(2)$.
  - It is LL(3) though.

LL(k) solves our problems?

- However, in general it does not work. For example, it is not hard to see that there does not exist any LL(k)-grammar accepting the following grammar.
  - S -> A | B
  - A -> a A b | ε
  - B -> a B b b | ε

Bottum up parsing



Example (LR parsing)

1: $S \to Ab$
2: $S \to Bb$
3: $A \to aa$
4: $B \to aaa$

Input = *aaab*

- We use 1 token lookahead, and use knowledge of everything we have seen so far.
- Initially, we only know that the input string starts with an *a*.
- This tells us nothing about whether we are in rule 1 or 2.
- We skip over the *a* , *remember it*, and look at the next token. This is another *a*.
- We have seen *aa*, which still tells us nothing about whether we are in rule 1 or 2.
- Again, we skip over the *a*, and *remember it*.
- We now *look* at the third token and find it is again an *a*.
- We now can decide whether the *aa* we saw earlier is derived from an A. As an *A* is always followed by a *b*, we decide this is not the case since the 3rd token is an *a*.

Example

1: $S \rightarrow Ab$
2: $S \rightarrow Bb$
3: $A \rightarrow aa$
4: $B \rightarrow aaa$

Input = *aaab*

- Reading the *a*, we know we are definitely in case 2 (assuming correct input).
- Our lookahead becomes a *b*, which is precisely what we expect. We therefore decide that the *aaa* we have seen so far results from a *B*.
- Note that at this point, we no longer need to 'remember' that we have seen *aaa*: all we need to know is that we've seen a *B*.
- This is effectively the same situation we would be in if we would have had *Bb* as input, and would have read *B*.
- So, it is enough to remember just that, which corresponds to a *reduce* action of an *LR* parser.
- Finally, we read the *b*, find that we are at the end of the input, decide that the lhs must be an *S* and decide that the input is accepted.

Leftmost and Rightmost derivations

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
E & \rightarrow & T \\
T & \rightarrow & \mathtt{id}
\end{array}
$$

Derivations for $\mathtt{id} + \mathtt{id}$:

$$
\begin{array}{lcl}
E & \Rightarrow & E + T \\
  & \Rightarrow & T + T \\
  & \Rightarrow & \mathtt{id} + T \\
  & \Rightarrow & \mathtt{id} + \mathtt{id}
\end{array}
$$

LEFTMOST

$$
\begin{array}{lcl}
E & \Rightarrow & E + T \\
  & \Rightarrow & E + \mathtt{id} \\
  & \Rightarrow & T + \mathtt{id} \\
  & \Rightarrow & \mathtt{id} + \mathtt{id}
\end{array}
$$

RIGHTMOST

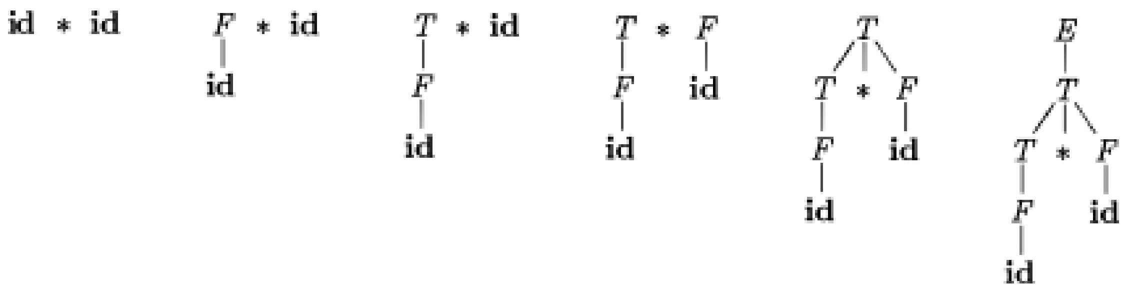
Bottom-up Parsing

- A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (bottom) and working up towards the root (top).

E -> E + T | T

T -> T * F | F

F -> (E) | id



Derivation in reverse:

E ⇒ T ⇒ T*F ⇒T*id ⇒ F*id ⇒ id*id

RIGHTMOST DERIVATION

Bottom-up Parsing

Given a stream of tokens *w*, *reduce* it to the start symbol.

$$E \rightarrow E+T$$
$$E \rightarrow T$$
$$T \rightarrow \mathrm{id}$$

Parse input stream id + id:

$$\mathrm{id} + \mathrm{id}$$
$$T + \mathrm{id}$$
$$E + \mathrm{id}$$
$$E + T$$
$$E$$

Reduction ≡ Derivation^{-1}

Bottom-Up LR

- Construct parse tree in a bottom-up manner
- Scan the input Left to right
- Find the Rightmost derivation in a reverse order
- Often more powerful than top-down parsing
  - Left recursion does not cause problems

Bottom-up Parsing

- Bottom-up parsing: the process of "reducing" the input string to the start symbol of the grammar.
- At each *reduction* step, a specific substring matching the rhs (body) of a production is replaced by the nonterminal at the lhs of this production.
- Key decisions to make:
  - when to reduce
  - which production rule to use.
- Use explicit parse stack
  - LR: on the stack is what has been *accepted*!
  - LL(k) : on the stack is what is *expected*!

Shift-Reduce parsing (example)

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow \mathtt{id}$

Left to Right Scan of input
Rightmost Derivation in reverse.

stack	input stream	action
$	id + id $	**shift**
$ id	+ id $	**reduce by** $T \rightarrow \mathtt{id}$
$ T	+ id $	**reduce by** $E \rightarrow T$
$ E	+ id $	**shift**
$ E +	id $	**shift**
$ E + id	$	**reduce by** $T \rightarrow \mathtt{id}$
$ E + T	$	reduce by $E \rightarrow E + T$
$ E	$	**ACCEPT**

Informal Example(1)

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

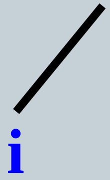
stack **tree** **input**

$

i + i $

shift

stack **tree** **input**



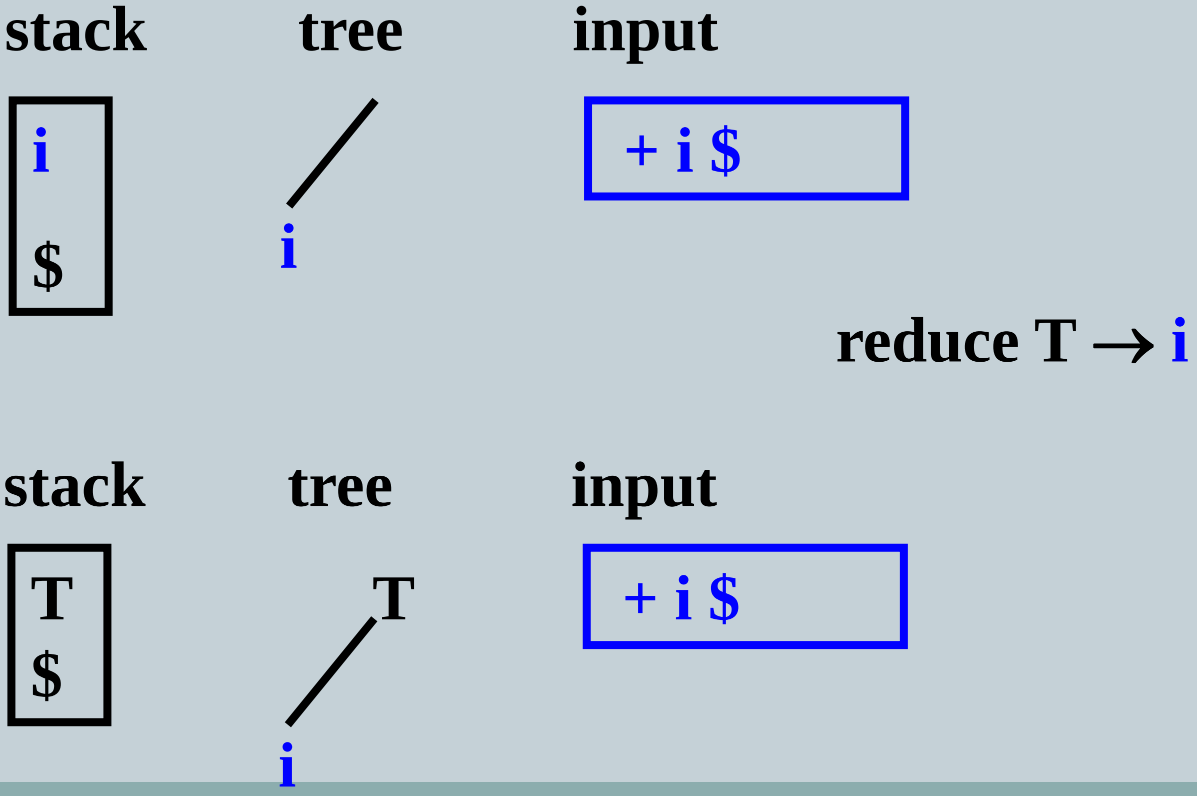
i

$

+ 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

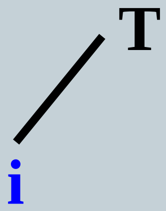
+ i $

Informal Example(3)

S → E $ E → T | E + T   T → i | (E)

stack **tree** **input**

T
$

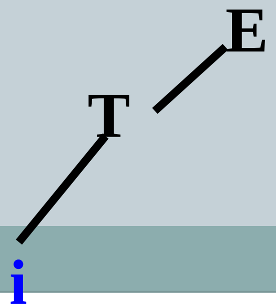


+ i $

reduce E → T

stack **tree** **input**

E
$



+ i $

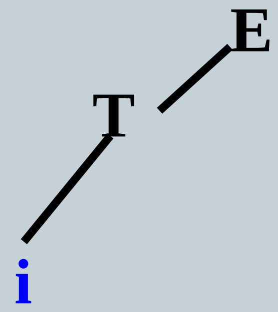
Informal Example(4)

S → E $ E → T | E + T   T → i | (E)

stack

E
$

tree



input

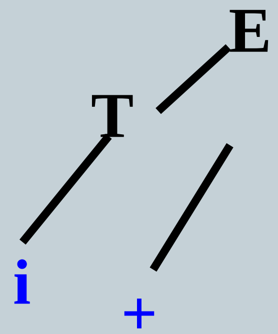
+ i $

shift

stack

+
E
$

tree

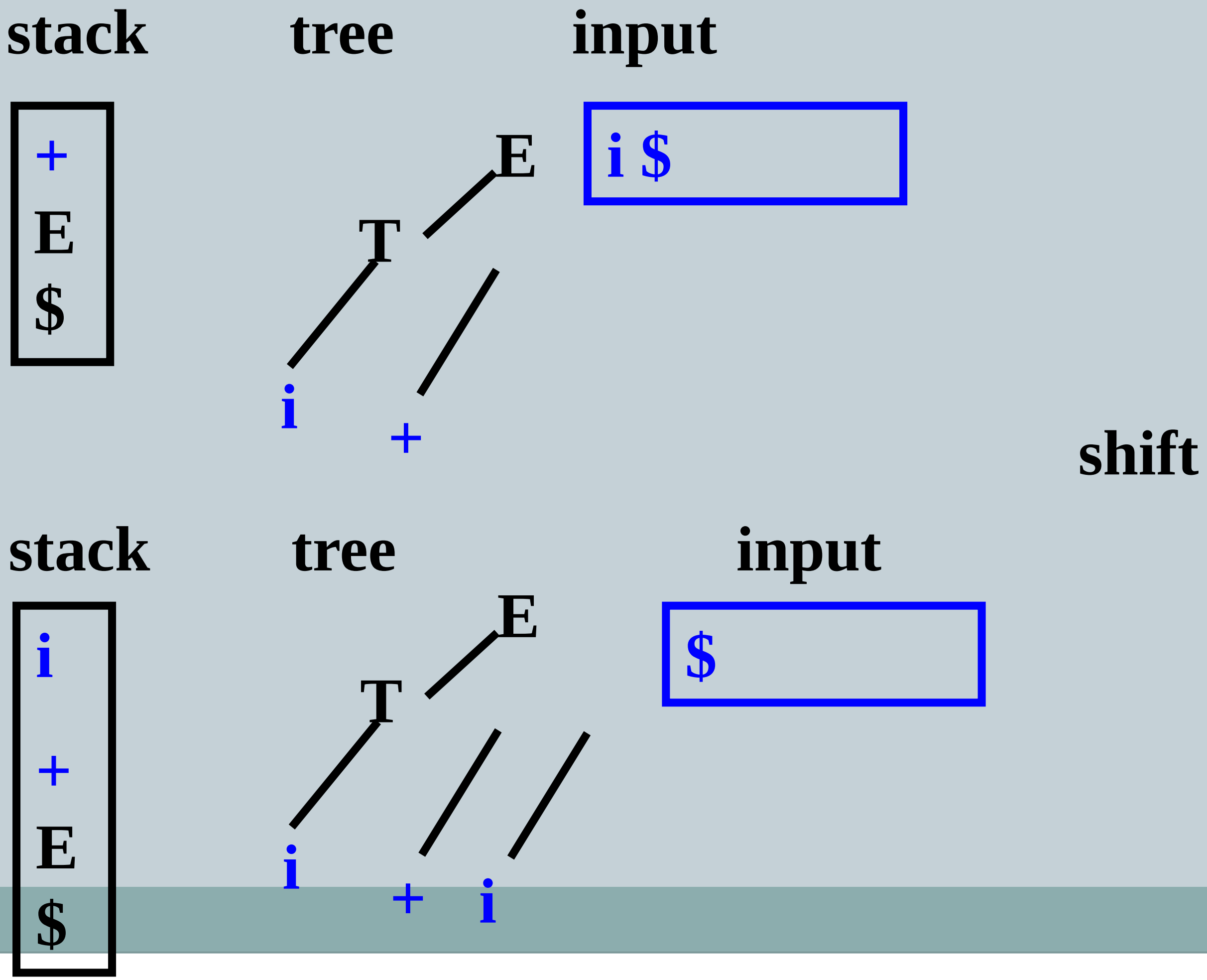


input

i $

Informal Example(5)

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



stack tree input

+
E
$

E
T
i
+

i $

shift

stack tree input

i
+
E
$

E
T
i
+
i

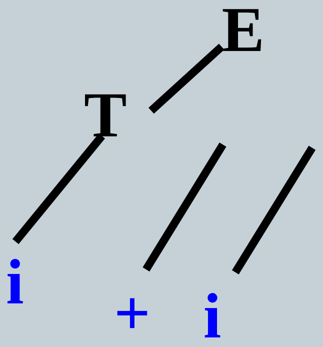
$

Informal Example(6)

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

stack tree input

i
+
E
$

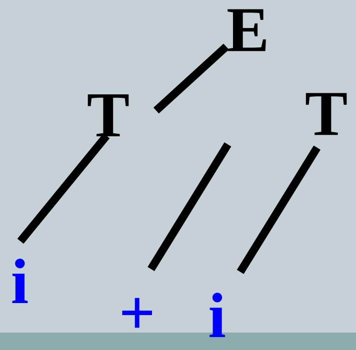


$

reduce $T \rightarrow i$

stack tree input

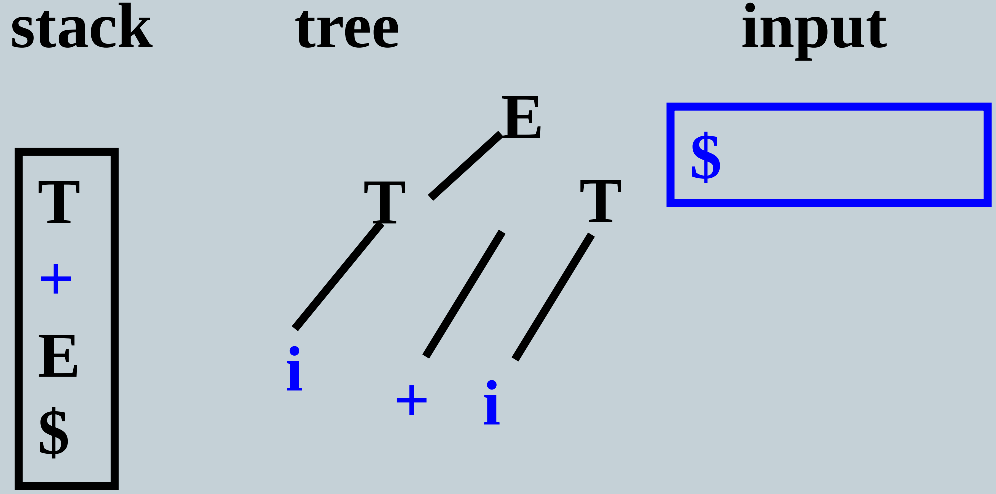
T
+
E
$



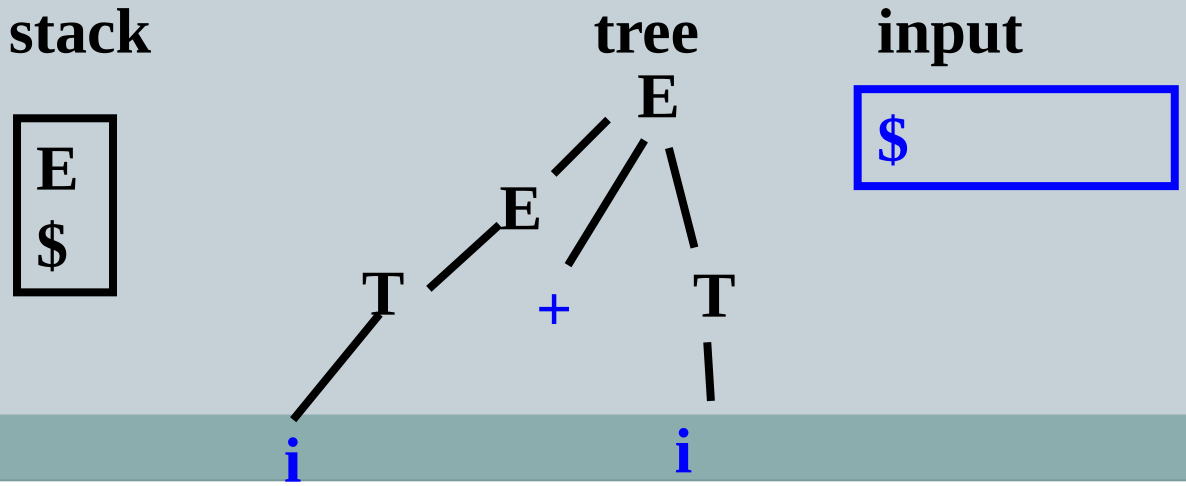
$

Informal Example(7)

S → E $ E → T | E + T   T → i | (E)



reduce E → E + T

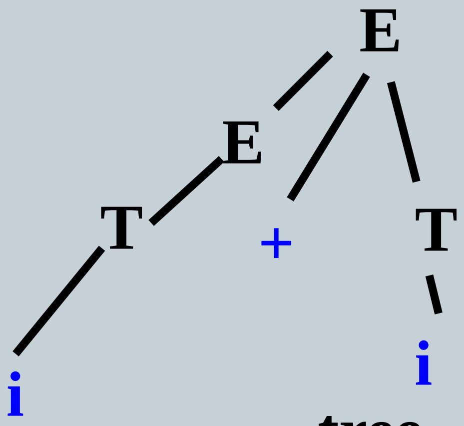


Informal Example(8)

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

stack **tree** **input**

E
$

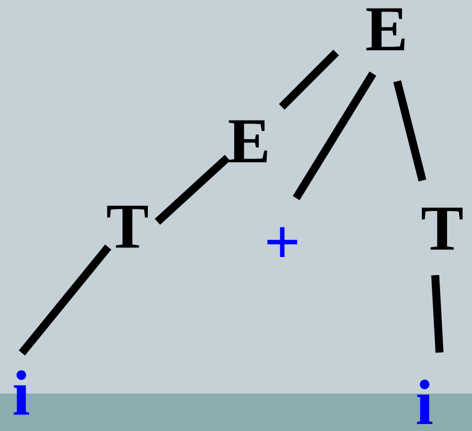


$

shift

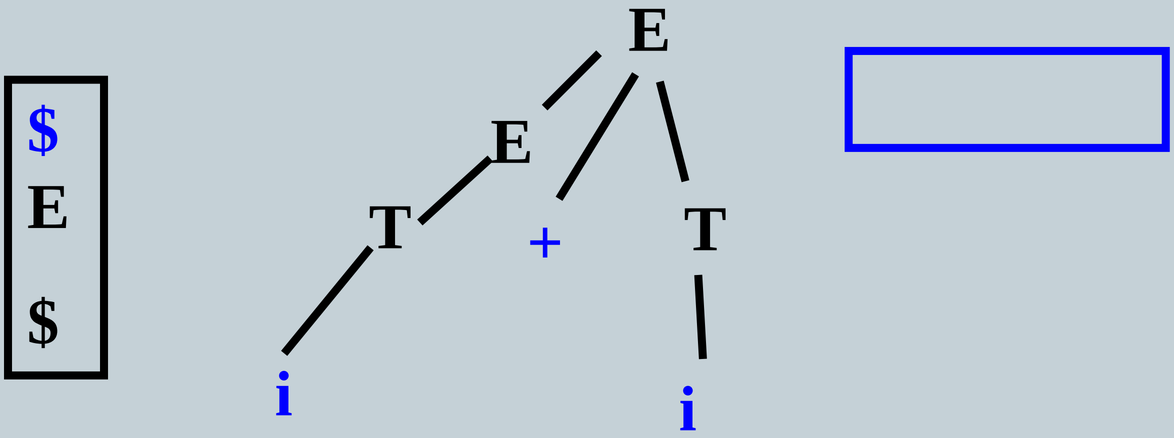
stack **tree** **input**

$
E
$



Informal Example(9)

stack **tree** **input**



reduce S → E $

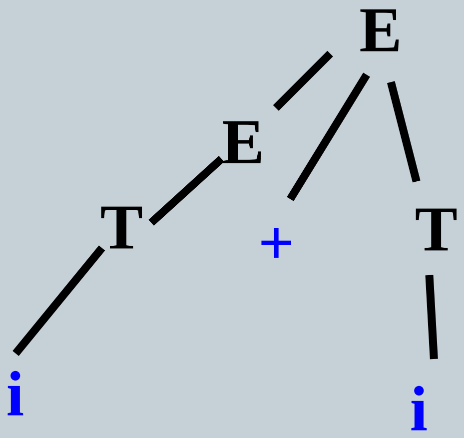
Informal Example(10)

stack

S

$

tree



input

ACCEPT

Informal Example

reduce S → E $

reduce E → E + T

reduce T → i

reduce E → T

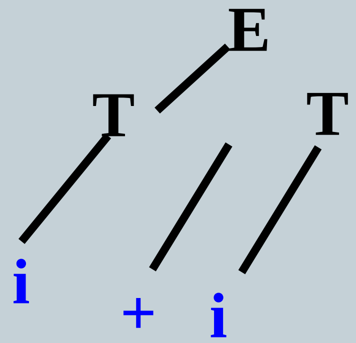
reduce T → i

Informal Example(7')

S → E $ E → T | E + T   T → i | (E)

stack **tree** **input**

T
+
E
$



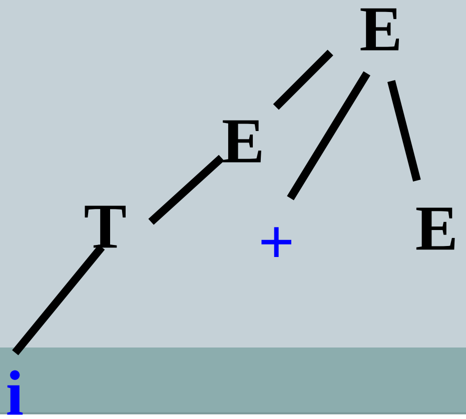
$

reduce E → T

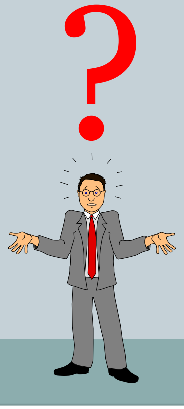
(was reduce E → E + T)

stack **tree** **input**

E
+
E
$



$



The Problems

Deciding between

- **shift and reduce**
- **reduce and reduce**

Even more complicated with epsilon rules

Grammar

S’ → S$
S → (S)S | ε

Stack	Input	Action
$	(( )) $	shift
$ (	( )) $	shift
$ ((	)) $	reduce S → ε
$ ((S	)) $	shift
$ ((S)	) $	reduce S → ε
$ ((S) S	) $	reduce S → (S) S
$ (S	) $	shift
$ (S)	$	reduce S → ε
$ (S) S	$	reduce S → (S) S
$ S	$	reduce
$ S’	$	ACCEPT

Reverse of rightmost derivation

(( ))
(( ))
(( ))
((S))
((S))
((S) S)
(S)
(S)
(S) S
S
S’

Shift-reduce parsers

- The parse stack contains symbols already parsed.
- The stack, concatenated with the remaining input always represents a *right sentential form*.
  - produced via a rightmost derivation
- Tokens are shifted onto the stack until the top of the stack contains a *handle*
- Then the handle is reduced by replacing it on the parse stack with the corresponding nonterminal.
- The decision whether to shift or to reduce is based on *a parse table*.
- The parsing is successful when the input has all been consumed and the stack contains only the goal symbol.

Shift-Reduce parsing

- Four operations:
  - **Shift:** Construct leftmost handle on top of stack
  - **Reduce:** Identify handle and replace by corresponding LHS
  - **Accept:** Stack reduced to start symbol and input token stream is empty
  - **Error:** Signal parse error if no handle is found.

Problem: How to identify a handle?

- Top of stack is the *rightmost* end of the handle. What is the leftmost end?

- If there are multiple productions with the handle on the RHS, which one to choose?

Solution:
Construct a parsing table, just as in the case of table driven LL(1) parsing.

A Simple Example of LR Parsing

$S \rightarrow B\,C$
$B \rightarrow \mathrm{a}$
$C \rightarrow \mathrm{a}$

Stack	Input Stream	Action
$	a a $	shift
$ a	a $	reduce by $B \rightarrow \mathrm{a}$
$ B	a $	shift
$ B a	$	reduce by $C \rightarrow \mathrm{a}$
$ B C	$	reduce by $S \rightarrow BC$
$ S	$	**ACCEPT**

Why did we reduce to B in step 2, while we reduced to C in step 4?

LR(0) Items

Item: A production with "•" somewhere on the RHS.

- Grammar symbols before the "•" are on stack;
- grammar symbols after the "•" represent symbols in the input stream.

input

$T \rightarrow \alpha \cdot \beta$

already matched α

expecting to match β

Item set: A set of items; corresponds to a state of the parser

A Simple Example of LR Parsing: a detailed look

Stack	Input	State	Action
\$	a a \$	$S' \to \bullet S$	

$S' \to S$

$S \to B\ C$

$B \to$ a

$C \to$ a

A Simple Example of LR Parsing: a detailed look

$S' \rightarrow S$
$S \rightarrow B\ C$
$B \rightarrow$ a
$C \rightarrow$ a

Stack	Input	State	Action
\$	a a \$	$S^{l} \rightarrow \bullet S$ $S \rightarrow \bullet BC$ $B \rightarrow \bullet$ a	

A Simple Example of LR Parsing: a detailed look

$S' \rightarrow S$
$S \rightarrow B\ C$
$B \rightarrow a$
$C \rightarrow a$

Stack	Input	State	Action
$	a a $	$S^{\prime} \rightarrow \bullet S$ $S \rightarrow \bullet BC$ $B \rightarrow \bullet a$	shift
$ a	a $		

A Simple Example of LR Parsing: a detailed look

$S' \to S$

$S \to B\ C$

$B \to a$

$C \to a$

Stack	Input	State	Action
\$	a a \$	$S' \to \bullet S$ $S \to \bullet BC$ $B \to \bullet a$	shift
\$ a	a \$	$B \to a \bullet$	

A Simple Example of LR Parsing: a detailed look

$S' \rightarrow S$
$S \rightarrow B\ C$
$B \rightarrow$ a
$C \rightarrow$ a

Stack	Input	State	Action
\$	a a \$	$S' \rightarrow \bullet S$ $S \rightarrow \bullet BC$ $B \rightarrow \bullet$ a	shift
\$ a	a \$	$B \rightarrow$ a$\bullet$	reduce by 3
\$ B	a \$		

A Simple Example of LR Parsing: a detailed look

$S' \rightarrow S$

$S \rightarrow B\ C$

$B \rightarrow a$

$C \rightarrow a$

Stack	Input	State	Action
$	a a $	$S' \rightarrow \bullet\ S$ $S \rightarrow \bullet\ BC$ $B \rightarrow \bullet\ a$	shift
$ a	a $	$B \rightarrow a\bullet$	reduce by 3
$ B	a $	$S \rightarrow B\bullet\ C$	

A Simple Example of LR Parsing: a detailed look

$S' \rightarrow S$

$S \rightarrow B\ C$

$B \rightarrow$ a

$C \rightarrow$ a

Stack	Input	State	Action
\$	a a \$	$S' \rightarrow \bullet S$ $S \rightarrow \bullet BC$ $B \rightarrow \bullet$ a	shift
\$ a	a \$	$B \rightarrow$ a$\bullet$	reduce by 3
\$ B	a \$	$S \rightarrow B \bullet C$ $C \rightarrow \bullet$ a	

A Simple Example of LR Parsing: a detailed look

$S' \rightarrow S$
$S \rightarrow B\ C$
$B \rightarrow a$
$C \rightarrow a$

Stack	Input	State	Action
$	a a $	$S' \rightarrow \bullet S$ $S \rightarrow \bullet BC$ $B \rightarrow \bullet a$	shift
$ a	a $	$B \rightarrow a \bullet$	reduce by 3
$ B	a $	$S \rightarrow B \bullet C$ $C \rightarrow \bullet a$	shift
$ B a	$		

A Simple Example of LR Parsing: a detailed look

$S' \rightarrow S$
$S \rightarrow B\ C$
$B \rightarrow \mathrm{a}$
$C \rightarrow \mathrm{a}$

Stack	Input	State	Action
$	a a $	$S' \rightarrow \bullet S$ $S \rightarrow \bullet BC$ $B \rightarrow \bullet \mathrm{a}$	shift
$ a	a $	$B \rightarrow \mathrm{a}\bullet$	reduce by 3
$ *B*	a $	$S \rightarrow B\bullet C$ $C \rightarrow \bullet \mathrm{a}$	shift
$ *B* a	$	$C \rightarrow \mathrm{a}\bullet$	

A Simple Example of LR Parsing: a detailed look

$S' \rightarrow S$

$S \rightarrow B\ C$

$B \rightarrow a$

$C \rightarrow a$

Stack	Input	State	Action
$\$$	$a\ a\ \$$	$S' \rightarrow \bullet\ S$ $S \rightarrow \bullet\ BC$ $B \rightarrow \bullet\ a$	shift
$\$\ a$	$a\ \$$	$B \rightarrow a\bullet$	reduce by 3
$\$\ B$	$a\ \$$	$S \rightarrow B\bullet\ C$ $C \rightarrow \bullet\ a$	shift
$\$\ B\ a$	$\$$	$C \rightarrow a\bullet$	reduce by 4
$\$\ B\ C$	$\$$		

A Simple Example of LR Parsing: a detailed look

$S' \rightarrow S$
$S \rightarrow B\ C$
$B \rightarrow \mathrm{a}$
$C \rightarrow \mathrm{a}$

Stack	Input	State	Action
\$	a a \$	$S' \rightarrow \cdot S$ $S \rightarrow \cdot BC$ $B \rightarrow \cdot \mathrm{a}$	shift
\$ a	a \$	$B \rightarrow \mathrm{a}\cdot$	reduce by 3
\$ B	a \$	$S \rightarrow B\cdot C$ $C \rightarrow \cdot \mathrm{a}$	shift
\$ B a	\$	$C \rightarrow \mathrm{a}\cdot$	reduce by 4
\$ B C	\$	$S \rightarrow BC\cdot$	

A Simple Example of LR Parsing: a detailed look

$S' \rightarrow S$

$S \rightarrow B\ C$

$B \rightarrow$ a

$C \rightarrow$ a

Stack	Input	State	Action
\$	a a \$	$S' \rightarrow \bullet\ S$ $S \rightarrow \bullet\ BC$ $B \rightarrow \bullet$ a	shift
\$ a	a \$	$B \rightarrow$ a$\bullet$	reduce by 3
\$ B	a \$	$S \rightarrow B\bullet\ C$ $C \rightarrow \bullet$ a	shift
\$ B a	\$	$C \rightarrow$ a$\bullet$	reduce by 4
\$ $B\ C$	\$	$S \rightarrow BC\bullet$	reduce by 2
\$ S	\$		

A Simple Example of LR Parsing: a detailed look

$S' \rightarrow S$

$S \rightarrow B\ C$

$B \rightarrow a$

$C \rightarrow a$

Stack	Input	State	Action
$	a a $	$S' \rightarrow \bullet S$ $S \rightarrow \bullet BC$ $B \rightarrow \bullet a$	shift
$ a	a $	$B \rightarrow a\bullet$	reduce by 3
$ B	a $	$S \rightarrow B\bullet C$ $C \rightarrow \bullet a$	shift
$ B a	$	$C \rightarrow a\bullet$	reduce by 4
$ B C	$	$S \rightarrow BC\bullet$	reduce by 2
$ S	$	$S' \rightarrow S\bullet$	

A Simple Example of LR Parsing: a detailed look

$S' \rightarrow S$

$S \rightarrow B\ C$

$B \rightarrow a$

$C \rightarrow a$

Stack	Input	State	Action
$\$$	$a\ a\ \$$	$S' \rightarrow \bullet\ S$ $S \rightarrow \bullet\ BC$ $B \rightarrow \bullet\ a$	shift
$\$\ a$	$a\ \$$	$B \rightarrow a\bullet$	reduce by 3
$\$\ B$	$a\ \$$	$S \rightarrow B\bullet\ C$ $C \rightarrow \bullet\ a$	shift
$\$\ B\ a$	$\$$	$C \rightarrow a\bullet$	reduce by 4
$\$\ B\ C$	$\$$	$S \rightarrow BC\bullet$	reduce by 2
$\$\ S$	$\$$	$S' \rightarrow S\bullet$	**ACCEPT**

LR parsing: another example

$E' \rightarrow E$
$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow \mathrm{id}$

Stack	Input	State	Action
$\$$	$\mathrm{id} + \mathrm{id}\ \$$	$E' \rightarrow \bullet E$ $E \rightarrow \bullet E+T$ $E \rightarrow \bullet T$ $T \rightarrow \bullet \mathrm{id}$	shift
$\$\ \mathrm{id}$	$+\ \mathrm{id}\ \$$	$T \rightarrow \mathrm{id}\bullet$	reduce by 4
$\$\ T$	$+\ \mathrm{id}\ \$$	$E \rightarrow T \bullet$	reduce by 3
$\$\ E$	$+\ \mathrm{id}\ \$$	$E' \rightarrow E\bullet$ $E \rightarrow E\bullet +T$	shift
$\$\ E\ +$	$\mathrm{id}\ \$$	$E \rightarrow E+\bullet T$ $T \rightarrow \bullet \mathrm{id}$	shift
$\$\ E\ +\ \mathrm{id}$	$\$$	$T \rightarrow \mathrm{id}\bullet$	reduce by 4
$\$\ E\ +\ T$	$\$$	$E \rightarrow E+T\bullet$	reduce by 2
$\$\ E$	$\$$	$E \rightarrow E\bullet +T$ $E' \rightarrow E\bullet$	**ACCEPT**

States of an LR parser

The states of an LR parser are sets of items

$E' \rightarrow E$
$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow \mathtt{id}$

I_0	$E' \rightarrow \cdot E$ $E \rightarrow \cdot E + T$ $E \rightarrow \cdot T$ $T \rightarrow \cdot \mathtt{id}$	Initial State $= closure(\{E' \rightarrow \cdot E\})$

Closure:

What other items are "equivalent" to a given item?

Given an item $A \rightarrow \alpha \cdot B\beta$, $closure(A \rightarrow \alpha \cdot B\beta)$ is the smallest set that contains

- the item $A \rightarrow \alpha \cdot B\beta$, and
- every item in $closure(B \rightarrow \cdot \gamma)$ for every production $B \rightarrow \gamma \in G$

States of an LR parser (contd.)

I_0	$E^{\prime} \rightarrow \cdot E$ $E \rightarrow \cdot E + T$ $E \rightarrow \cdot T$ $T \rightarrow \cdot \mathtt{id}$	Initial State $= closure(\{E^{\prime} \rightarrow \cdot E\})$
I_3	$T \rightarrow \mathtt{id} \cdot$	$= goto(I_0, \mathtt{id})$

Goto:

$goto(I, X)$ specifies the next state to visit.

X is a terminal: when the next symbol on input stream is X.

X is a nonterminal: when the last reduction was to X.

$goto(I, X)$ contains all items in $closure(A \rightarrow \alpha X \cdot \beta)$ for every item $A \rightarrow \alpha \cdot X \beta \in I$.

State sets construction

$E' \rightarrow E$		$E \rightarrow T$	
$E \rightarrow E + T$		$T \rightarrow \mathrm{id}$	

I_0	$= closure(\{E' \rightarrow \bullet E\})$	$E' \rightarrow \bullet E$ $E \rightarrow \bullet E + T$ $E \rightarrow \bullet T$ $T \rightarrow \bullet \mathrm{id}$
I_1	$= goto(I_0, E)$	$E' \rightarrow E \bullet$ $E \rightarrow E \bullet + T$
I_2	$= goto(I_0, T)$	$E \rightarrow T \bullet$
I_3	$= goto(I_0, \mathrm{id})$	$T \rightarrow \mathrm{id} \bullet$
I_4	$= goto(I_1, +)$	$E \rightarrow E + \bullet T$ $T \rightarrow \bullet \mathrm{id}$
I_5	$= goto(I_4, T)$	$E \rightarrow E + T \bullet$

LR Action and goto tables

$1:\ E' \rightarrow E$ $\quad$ $3:\ E \rightarrow T$

$2:\ E \rightarrow E + T$ $\quad$ $4:\ T \rightarrow \text{id}$

I_0	$= closure(\{E' \rightarrow \bullet E\})$	$E' \rightarrow \bullet E$ $E \rightarrow \bullet E + T$ $E \rightarrow \bullet T$ $T \rightarrow \bullet \text{id}$
I_1	$= goto(I_0, E)$	$E' \rightarrow E \bullet$ $E \rightarrow E \bullet + T$
I_2	$= goto(I_0, T)$	$E \rightarrow T \bullet$
I_3	$= goto(I_0, \text{id})$	$T \rightarrow \text{id} \bullet$
I_4	$= goto(I_1, +)$	$E \rightarrow E + \bullet T$ $T \rightarrow \bullet \text{id}$
I_5	$= goto(I_4, T)$	$E \rightarrow E + T \bullet$

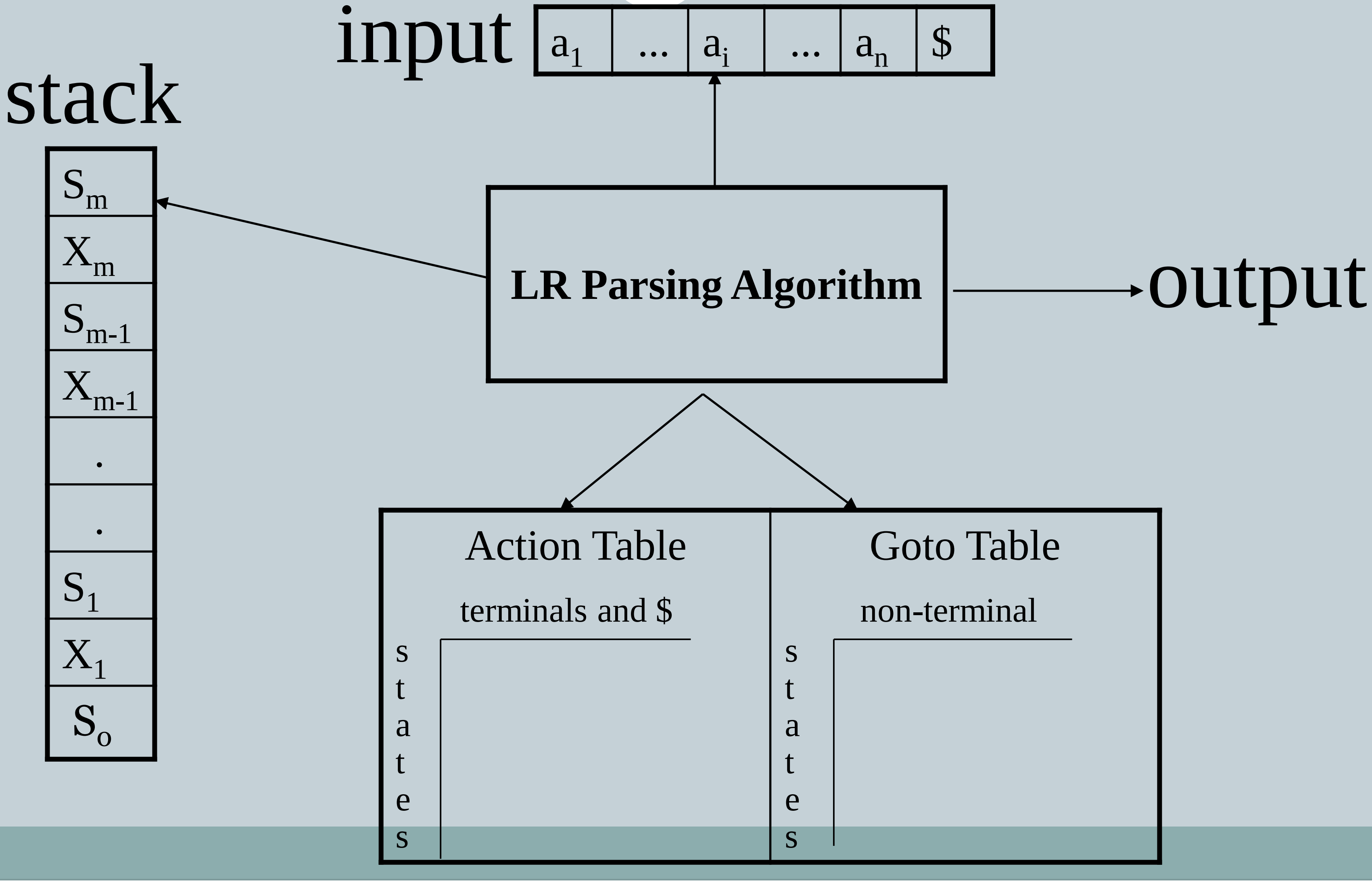
Action table

	id	+	$
0	S, 3		
1		S, 4	Acc
2	R3	R3	R3
3	R4	R4	R4
4	S, 3		
5	R2	R2	R2

Goto table

	E	T
0	1	2
1		
2		
3		
4		5
5		

LR parsing push down automaton



LR parsing algorithm

```
Loop forever
  switch action(state stack.top(), current token)
    case shift $s'$:
      symbol stack.push(current token);
      state stack.push($s'$);
      next token();
    case reduce $A \rightarrow \beta$:
      pop $|\beta|$ symbols from symbol stack;
      symbol stack.push($A$);
      pop state stack;
      state stack.push(goto(state stack.top(), $A$));
    case accept: return;
    default: error;
```

Final LR parser: states and transitions

Action Table:

	id	+	$
0	S, 3		
1		S, 4	A
2	$R3$	$R3$	$R3$
3	$R4$	$R4$	$R4$
4	S, 3		
5	$R2$	$R2$	$R2$

Goto Table:

	E	T
0	1	2
1		
2		
3		
4		5
5		

1: $E' \rightarrow E$ 3: $E \rightarrow T$
2: $E \rightarrow E + T$ 4: $T \rightarrow$ id

State stack	Symbol stack	Input	Action
$ 0	$	id + id $	shift, 3
$ 0 3	$ id	+ id $	reduce by 4
$ 0 2	$ T	+ id $	reduce by 3
$ 0 1	$ E	+ id $	shift, 4
$ 0 1 4	$ E +	id $	shift, 3
$ 0 1 4 3	$ E + id	$	reduce by 4
$ 0 1 4 5	$ E + T	$	reduce by 2
$ 0 1	$ E	$	**ACCEPT**

LR parsing summary

Table-driven shift reduce parsing:

Shift	Move **terminal** symbols from input stream to stack.
Reduce	Replace top elements of stack that form an instance of the RHS of a production with the corresponding LHS
Accept	Stack top is the start symbol when the input stream is exhausted

Table constructed using LR(0) Item Sets.

Try yourself!

S → E $

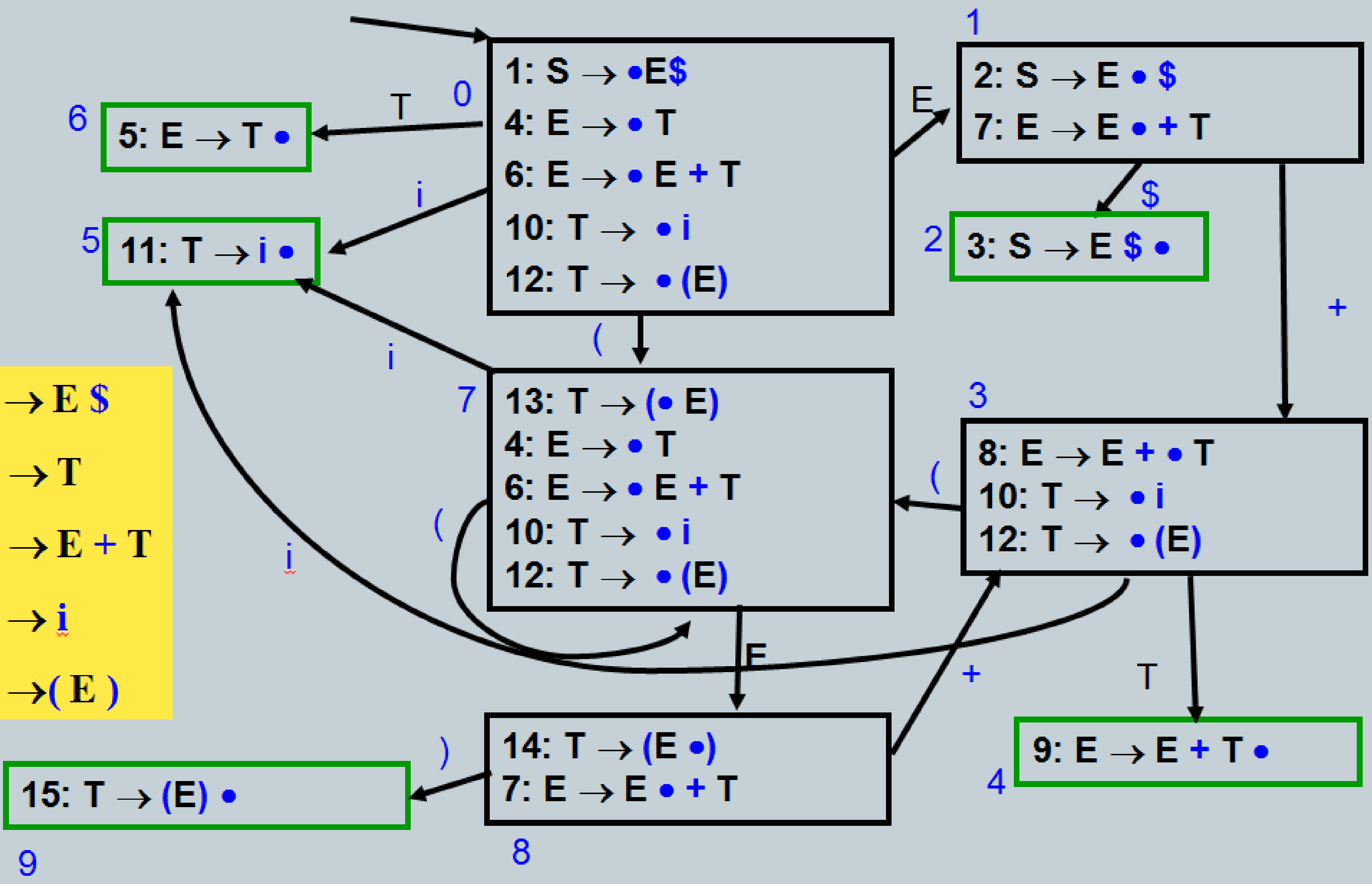
E → T

E → E + T

T → i

T → (E)

LR(0) items	(	)	i	+	$	T	E	Reduce
1: S → • E $ **E → • T** **E → • E + T** **T → • i** **T → • (E)**	s5		s4			3	2	
2: S → E • $ **E → E • + T**				s6	Acc			
3: E → T •								r
4: T → i •								r
5: T → (• E) **E → • T** **E → • E + T** **T → • i** **T → • (E)**	s5		s4			3	7	
6: E → E + • T **T → • i** **T → • (E)**	s5		s4			8		
7: T → (E •) **E → E • + T**		s9		s6				
8: E → E + T •								r
9: 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

3: S → E $ •

5: E → T •

11: T → i •

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)

9: E → E + T •

14: T → (E •)
7: E → E • + T

15: T → (E) •

S → E $

E → T

E → E + T

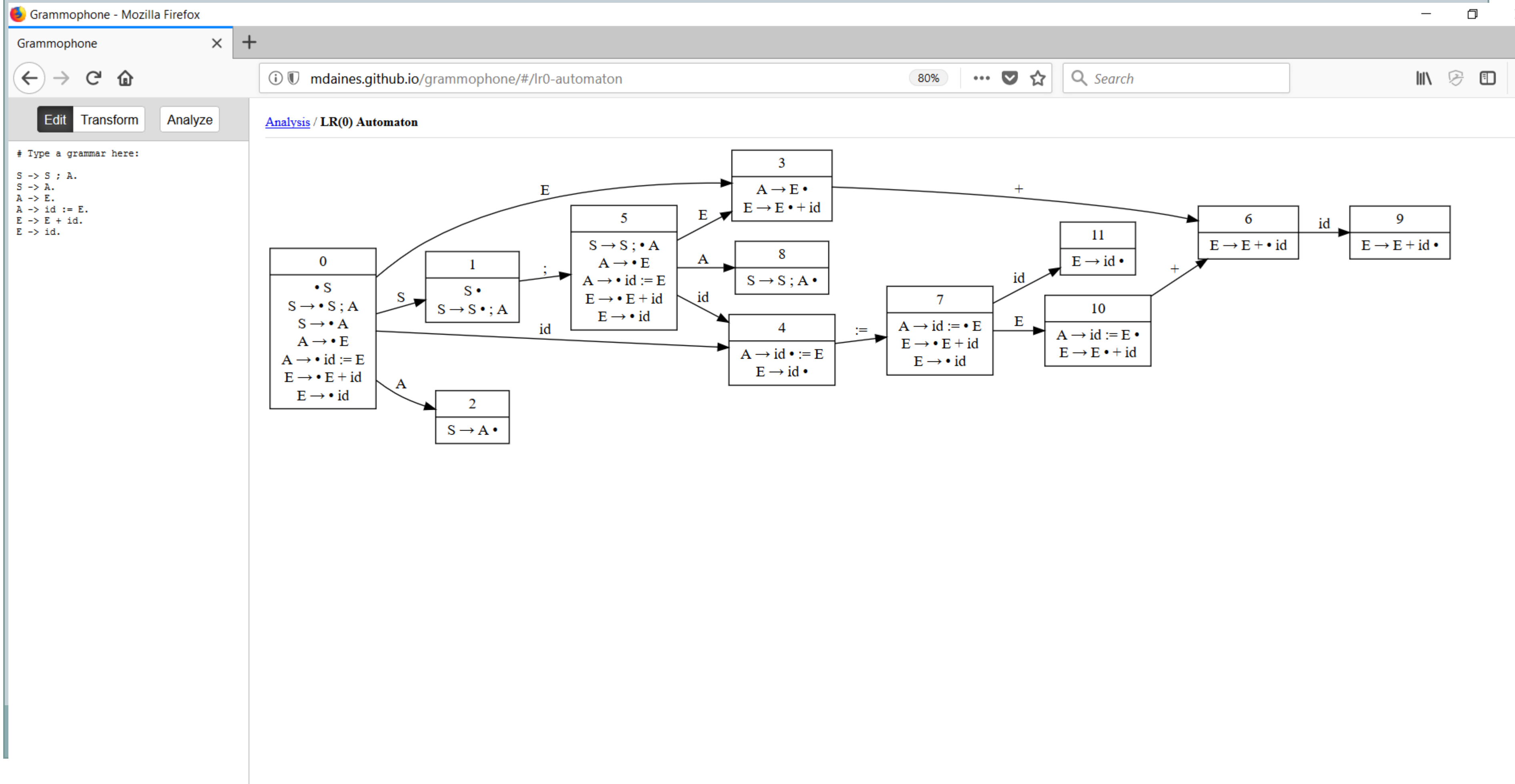
T → i

T → (E)

Warning: states have been renumbered in comparison with previous slide!

Nice web site for grammar analysis

http://mdaines.github.io/grammophone



```
# Type a grammar here:

S -> S ; A.
S -> A.
A -> E.
A -> id := E.
E -> E + id.
E -> id.
```

Conflicts in Parsing table

Grammar:

Grammar:

1 : S S’ →→ a S S$

2 : S → a S

3 : S →

$I_0 = closure(\{S' \to \bullet S \$\})$:

$S' \to \bullet S \$$

$S \to \bullet \mathrm{a} S$

$S \to \bullet$

$I_1 = \mathrm{goto}(I_0, \mathrm{S})$:

$S' \to S \bullet \$$

$I_2 = \mathrm{goto}(I_0, \mathrm{a})$:

$S \to \mathrm{a} \bullet S$

$S \to \bullet \mathrm{a} S$

$S \to \bullet$

$I_3 = \mathrm{goto}(I_2, \mathrm{S})$:

$S \to \mathrm{a} S \bullet$

Action Table:

	a	$
0	S, 2 R 3	R 3
1		Accept
2	S, 2 R 3	R 3
3	R 2	R 2

2 Shift-Reduce Conflicts

Idea: Choose **shift** because a is not in Follow(S)

SLR(1) Parsing

SLR(1) parsing makes a reduction by $A \rightarrow \alpha$ in state i if the current token is **a** and:

$A \rightarrow \alpha.$ in I_i

a is in Follow(A)

Simple LR (SLR) Parsing

Construct Action Table *action*, indexed by *states* × *terminals*, and Goto Table *goto*, indexed by *states* × *nonterminals*:

Construct $\{I_0, I_1, \ldots, I_n\}$, the set of LR(0) item sets of the grammar. For each i, $0 \le i \le n$, do the following:

- If $A \rightarrow \alpha \cdot a\beta \in I_i$ and $goto(I_i, a) = I_j$ then set $action[i, a] = shift\ \ j$
- If $A \rightarrow \gamma \cdot \in I_i$ (A is not the start symbol) then
  for each $a \in FOLLOW(A)$, set $action[i, a] = reduce\ \ A \rightarrow \gamma$
- If $S' \rightarrow S \cdot \$ \in I_i$ then set $action[i, \$] = accept$
- If $goto(I_i, A) = I_j$ (A is a nonterminal) then set $goto[i, A] = j$

SLR(1) parsing table

$S' \rightarrow S\ \$$
$S \rightarrow a\ S$
$S \rightarrow$

	a	$
0	S, 2 R 3	R 3
1		Accept
2	S, 2 R 3	R 3
3	R 2	R 2

Item Sets:

I_0	$= closure(\{S' \rightarrow \bullet\ S\})$	$S' \rightarrow \bullet\ S$ $S \rightarrow \bullet\ a\ S$ $S \rightarrow \bullet$
I_1	$= goto(I_0, S)$	$S' \rightarrow S\bullet$
I_2	$= goto(I_0, a)$	$S \rightarrow a\ \bullet\ S$ $S \rightarrow \bullet\ a\ S$ $S \rightarrow \bullet$
I_3	$= goto(I_2, S)$	$S \rightarrow a\ S\ \bullet$

$FOLLOW(S) = \{\$\}$

SLR Action Table:

	a	$
0	S, 2	R 3
1		Acc
2	S, 2	R 3
3		R 2

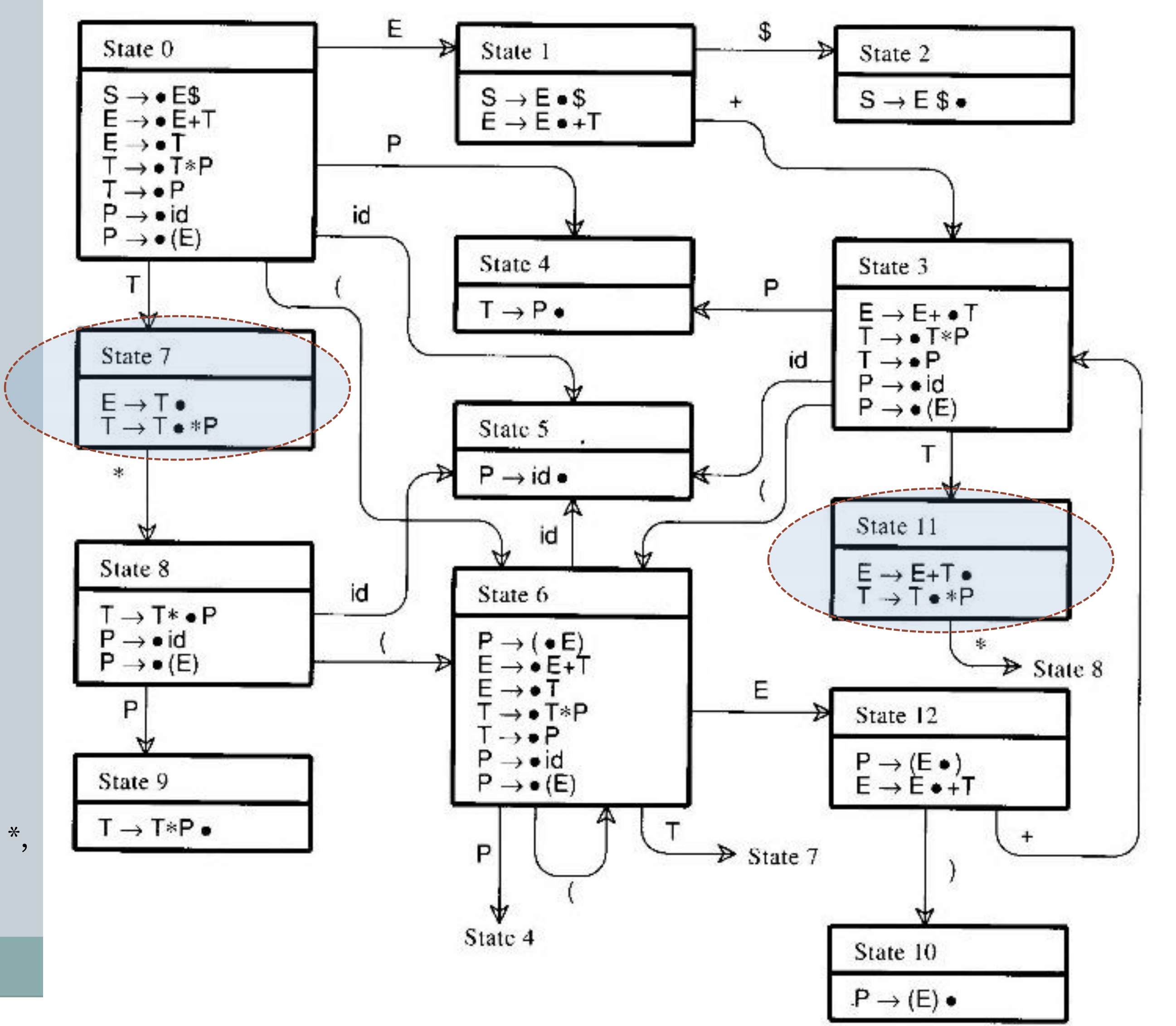
Another example: SLR(1) Parsing

S→E $
E→E + T | T
T→T * P | P
P→id | (E)

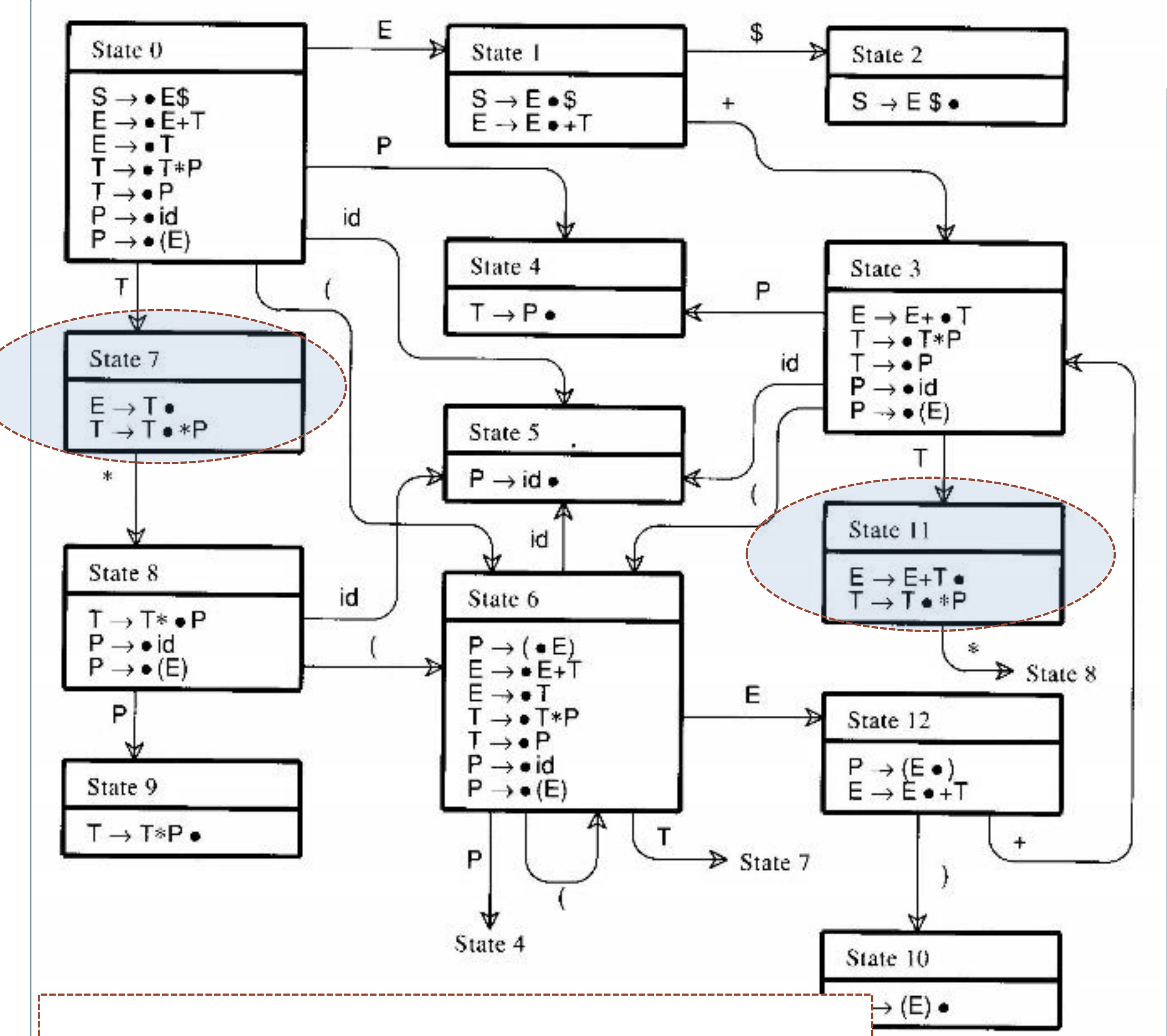
Is this an LR(0) Grammar?

The grammar is not LR(0):
See states 7,11
Follow(E)={$,+,)}

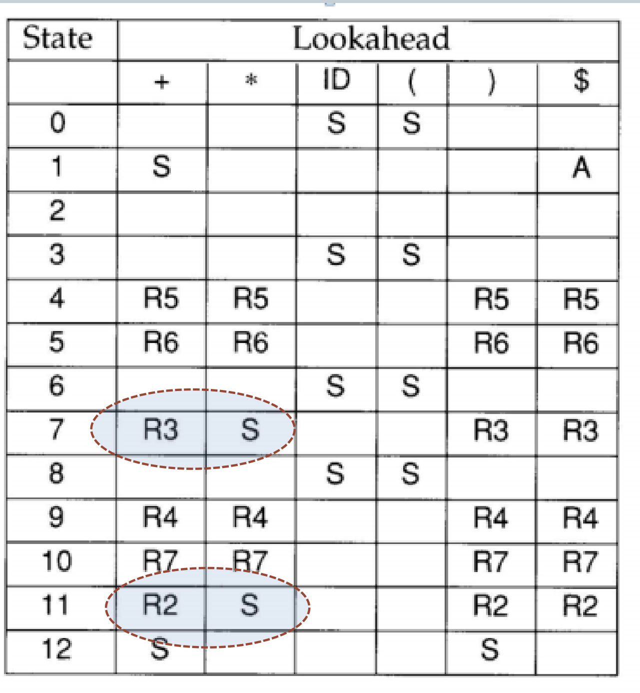
So, the grammar is SLR(1):
In these states, shift if the lookahead is *, otherwise reduce.



Example: SLR(1) Parsing



S→E $
E→E + T | T
T→T * P | P
P→id | (E)



State	Lookahead					
	+	*	ID	(	)	$
0			S	S		
1	S					A
2						
3			S	S		
4	R5	R5			R5	R5
5	R6	R6			R6	R6
6			S	S		
7	R3	S			R3	R3
8			S	S		
9	R4	R4			R4	R4
10	R7	R7			R7	R7
11	R2	S			R2	R2
12	S				S	

The grammar is SLR(1)

Exercise

Consider the following grammar:

0: S → E

1: E → 1 E

2: E → 1

Show that the grammar can be parsed by an SLR parser but not by an LR(0) parser.