# Topic 12 – Bottom-up Parsing

**Key Ideas**

- LR Parsing
- shifting, reducing
- using a transducer to parse
- shift-reduce and reduce-reduce conflicts

**References**

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 3.14- 3.15

# Non- LL(1) Grammars

**A Non-LL(1) Grammar**

G: $L = \{a^n b^m \mid n \geq m \geq 0\}$

- i.e. the number of a's is greater or equal to the number of b's
- L is not LL(*k*) for any *k:* just make the run of a's larger than *k*

**Ambiguous Version of Grammar**

$G_1$: $S \rightarrow \varepsilon$
$S \rightarrow aS$
$S \rightarrow aSb$

**Unambiguous Version of Grammar**

$$\left.\begin{array}{lll} G_2: S \rightarrow \vdash A \dashv & & B \rightarrow \varepsilon \\ \phantom{G_2:} A \rightarrow aA & & B \rightarrow aBb \\ \phantom{G_2:} A \rightarrow B & & \end{array}\right\} \begin{array}{l}\text{A generates excess a's} \\ \text{B generates pairs of a's and b's}\end{array}$$

# LR Parsing

**LL vs. LR**

- Recall that a stack in LL/top-down parsing is used in the following way:
  - the derivation progresses from the top of the parse tree (S’) down to the bottom, i.e. a *top-down derivation*
  - current step in derivation = *input processed + stack*
  - the the stack is read from *top to bottom*
- For LR/bottom-up parsing, we have
  - the derivation progresses from the bottom of the parse tree up to the top (i.e. S’), i.e. a *bottom-up derivation*
  - current step in derivation: *stack + input to be read*
  - stack is read from *bottom to top*

# Sample CFG

**Sample Grammar**

- Recall our Augmented Grammar

    1. S’ $\rightarrow$ ⊢ S ⊣
    2. S $\rightarrow$ AyB
    3. A $\rightarrow$ ab
    4. A $\rightarrow$ cd
    5. B $\rightarrow$ z
    6. B $\rightarrow$ wz

- LL parsing is intuitive: *read* from the *left, parse* from the *left*
- For *LR parsing, read* from the *left* and *parse* from the *right*
    - i.e. parse using a rightmost derivation

# Recall: Example of *LL(1) Parsing*

## LL(1) Parsing

| | Derivation | Read | Input | Stack | Action |
|---|---|---|---|---|---|
| 1 | S’ | | ⊢ abywz ⊣ | > S’ | expand (1) |
| 2 | ⊢ S ⊣ | | **⊢** abywz ⊣ | > **⊢** S ⊣ | match |
| 3 | ⊢ S ⊣ | ⊢ | abywz ⊣ | > **S** ⊣ | expand (2) |
| 4 | ⊢ AyB ⊣ | ⊢ | abywz ⊣ | > **A** y B ⊣ | expand (3) |
| 5 | ⊢ AyB ⊣ | ⊢ | **a**bywz ⊣ | > **a** b y B ⊣ | match |
| 6 | ⊢ abyB ⊣ | ⊢ a | **b**ywz ⊣ | > **b** y B ⊣ | match |
| 7 | ⊢ abyB ⊣ | ⊢ ab | **y**wz ⊣ | > **y** B ⊣ | match |
| 8 | ⊢ abyB ⊣ | ⊢ aby | wz ⊣ | > **B** ⊣ | expand (6) |
| 9 | ⊢ abywz ⊣ | ⊢ aby | **w**z ⊣ | > **w** z ⊣ | match |
| 10 | ⊢ abywz ⊣ | ⊢ abyw | **z** ⊣ | > **z** ⊣ | match |
| 11 | ⊢ abywz ⊣ | ⊢ abywz | **⊣** | > **⊣** | ACCEPT |

# Example of *LR Parsing*

## LR Parsing

| | Derivation | Stack | Read | Input | Action |
|---|---|---|---|---|---|
| 1 | ⊢ abywz ⊣ | ⊢ < | ⊢ | abywz⊣ | shift ⊢ |
| 2 | ⊢ abywz ⊣ | ⊢ a < | ⊢ a | bywz⊣ | shift a |
| 3 | ⊢ abywz ⊣ | ⊢ ab < | ⊢ ab | ywz⊣ | shift b |
| 4 | ⊢ Aywz ⊣ | ⊢ A < | ⊢ ab | ywz⊣ | reduce (3) |
| 5 | ⊢ Aywz ⊣ | ⊢ Ay < | ⊢ aby | wz⊣ | shift y |
| 6 | ⊢ Aywz ⊣ | ⊢ Ayw < | ⊢ abyw | z⊣ | shift w |
| 7 | ⊢ Aywz ⊣ | ⊢ Aywz < | ⊢ abywz | ⊣ | shift z |
| 8 | ⊢ AyB ⊣ | ⊢ AyB < | ⊢ abywz | ⊣ | reduce(6) |
| 9 | ⊢ S ⊣ | ⊢ S < | ⊢ abywz | ⊣ | reduce (2) |
| 10 | ⊢ S ⊣ | ⊢ S ⊣ < | ⊢ abywz ⊣ | ε | shift ⊣ |
| 11 | S’ | S’ < | ⊢ abywz ⊣ | ε | reduce (1) |

# Comparing LL vs. LR Parsing

**LL vs. LR**

- *Derivation Column*
  - in LL: it goes from S’ to ⊢ abywz ⊣ (i.e. down the parse tree)
  - in LR: it goes from ⊢ abywz ⊣ to S’ (i.e. up the parse tree)
- *Top of the Stack*
  - in LL: the top of the stack is on the left when we read it
  - in LR: the top of the stack is on the right when we read it
- *Terminals in the Stack*
  - in LL: at one stage, the stack had many of the terminals from the beginning of the input on the stack: > a b y B ⊣
  - in LR: at one stage the stack had many of the terminals from the end of the input on the stack: ⊢ A y w z <

# LR Parsing

**LR Operations**

There are two operations in LR Parsing

1. *Shift*
   - move a character from the input file to the stack
   - we'll also include it in the "Read" column to keep track of what has been read so far.
2. *Reduce*
   - If there is a production rule of the form $S \rightarrow AyB$ and $AyB$ is on the stack then reduce (i.e. replace) $AyB$ to $S$
   - this step is the act of applying a production rule to simplify what is on the stack

# Bottom-Up Parsing

## Parsing the Input

- To start, keep on shifting input onto the stack until you have a match with the right hand side (RHS) of some production rule.

| | Derivation | Stack | Read | Input | Action |
|---|---|---|---|---|---|
| 1 | ⊢ abywz ⊣ | ⊢ < | ⊢ | abywz⊣ | shift ⊢ |
| 2 | ⊢ abywz ⊣ | ⊢ a < | ⊢ a | bywz⊣ | shift a |
| 3 | ⊢ abywz ⊣ | ⊢ ab < | ⊢ ab | ywz⊣ | shift b |

- Now there is a match between the stack and the RHS of rule 3, $A \rightarrow ab$, so reduce (i.e. replace) what is on the stack, ab, to the left hand side (LHS) of that same rule, i.e. A.

| | | | | | |
|---|---|---|---|---|---|
| 4 | ⊢ Aywz ⊣ | ⊢ A < | ⊢ ab | ywz⊣ | reduce (3) |

# Bottom-Up Parsing

**Parsing the Input**

- Again, keep on shifting input onto the stack until you have a match with the RHS of some production rule.

| | Derivation | Stack | Read | Input | Action |
|---|---|---|---|---|---|
| 5 | ⊢ Aywz ⊣ | ⊢ A y < | ⊢ aby | wz⊣ | shift y |
| 6 | ⊢ Aywz ⊣ | ⊢ Ayw < | ⊢ abyw | z⊣ | shift w |
| 7 | ⊢ Aywz ⊣ | ⊢ Aywz < | ⊢ abywz | ⊣ | shift z |

- Now there is a match between what is on the top of the stack and the RHS of rule 6, B → wz, so reduce wz to the LHS of that same rule, i.e. B.

| | | | | | |
|---|---|---|---|---|---|
| 8 | ⊢ AyB ⊣ | ⊢ AyB < | ⊢ abywz | ⊣ | reduce(6) |

# Bottom-Up Parsing

## Parsing the Input

- After that reduction there is yet another match with the RHS of a production rule, so there is no need to shift.

| | Derivation | Stack | Read | Input | Action |
|---|---|---|---|---|---|
| 8 | ⊢ AyB ⊣ | ⊢ AyB < | ⊢ abywz | ⊣ | reduce(6) |

- There is a match between what is on the top of the stack and the RHS of rule 2, S $\rightarrow$ AyB, so reduce AyB to the LHS of that same rule, i.e. S.

| | | | | | |
|---|---|---|---|---|---|
| 9 | ⊢ S ⊣ | ⊢ S < | ⊢ abywz | ⊣ | reduce (2) |

# Bottom-Up Parsing

## Parsing the Input

- Again, keep on shifting input onto the stack until you have a match with the RHS of some production rule.

| | Derivation | Stack | Read | Input | Action |
|---|---|---|---|---|---|
| 10 | ⊢ S ⊣ | ⊢ S ⊣ < | ⊢ abywz ⊣ | ε | shift ⊣ |

- There is a match between what is on the stack and the RHS of rule 1, S’ → ⊢ S ⊣, so reduce ⊢ S ⊣ to S’.

| | | | | | |
|---|---|---|---|---|---|
| 11 | S’ | S’ < | ⊢ abywz ⊣ | ε | reduce (1) |

- The start symbol, S’, is now the only symbol on the stack so the input ⊢ abywz ⊣ has been derived from S’ and is a string in the language generated by the grammar.

# Shift / Reduce

**When to Shift, When to Reduce**

- *Key Question:* How do you know when to shift and when to reduce?
  - for LL(1) parsing, we have a predictor table
  - for LR(1) parsing, we have a transducer
    - i.e. a DFA that recognizes strings and may produce output during a transition from one state to another
    - you will need to review / recall transducers for A7
- In 1965 Donald Knuth proved a theorem that we can construct a DFA (really, a transducer) for LR(1) grammars

# Shift / Reduce

**When to Shift, When to Reduce**

- *Key Question:* How do you know when to shift and when to reduce?
- *Key Idea:* Introduce the symbol "•" as a place holder to help us keep track of where we are in the RHS of a production rule,

  e.g. $S' \rightarrow \bullet \vdash E \dashv$

  $S' \rightarrow \vdash \bullet E \dashv$

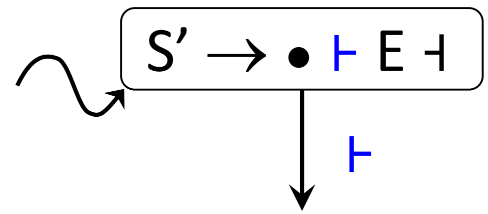
  $S' \rightarrow \vdash E \bullet \dashv$

  $S' \rightarrow \vdash E \dashv \bullet$
- We *create a finite automaton* to track the progress of the placeholder through the various production rules
- There is a different state each time the place holder moves over one symbol in the production rule

# Building an LR(0) automaton

**Sample Grammar**

G: 1. $S' \rightarrow \vdash E \dashv$
2. $E \rightarrow E + T$
3. $E \rightarrow T$
4. $T \rightarrow id$

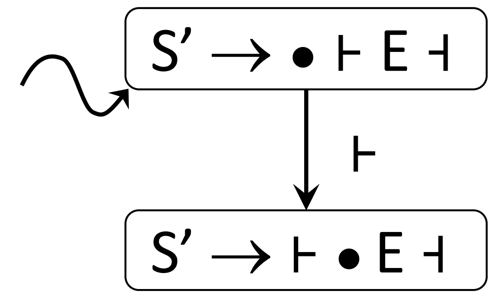



- *Start state:* make the start state the first rule, with a dot (•) in front of the leftmost symbol of the RHS, e.g. $S' \rightarrow \bullet \vdash E \dashv$
- *For each state:* create a transition out of that state with the symbol that follows the “•”
- Here the BOF symbol “⊢” follows the “•” so have a transition out of the start state labelled ⊢.

# Building an LR(0) automaton

**Sample Grammar**

G: 1. $S' \to \vdash E \dashv$
2. $E \to E + T$
3. $E \to T$
4. $T \to id$

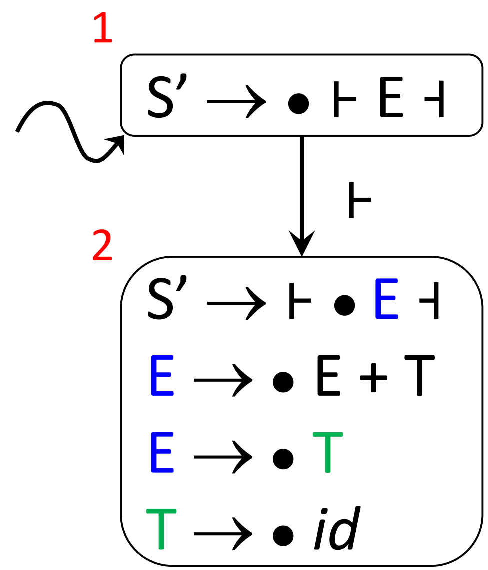



- Here the RHS of the start state is "$\bullet \vdash E \dashv$"
- Advancing the "$\bullet$" forward by one symbol creates the new state "$S' \to \vdash \bullet E \dashv$"
- This transition is saying with input $\vdash$ the automaton will advance from state "$S' \to \bullet \vdash E \dashv$" to state "$S' \to \vdash \bullet E \dashv$"
- A rule with a "$\bullet$" somewhere on the RHS is called an *item*. It indicates a partially completed rule.

# Building an LR(0) automaton

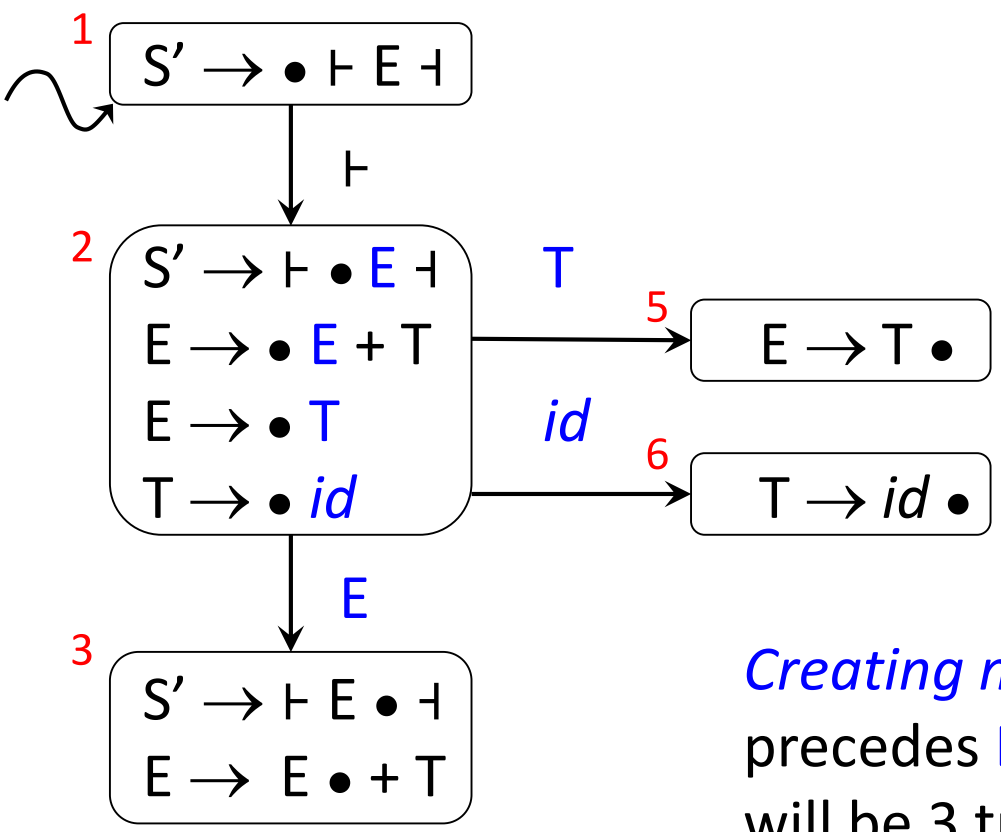
**Sample Grammar**

G: 1. $S' \to \vdash E \dashv$
2. $E \to E + T$
3. $E \to T$
4. $T \to id$




- *For non-terminals:* If "•" precedes a non-terminal (in this case E) add all productions with that non-terminal on the LHS to the current state (and place the "•" in the leftmost position of those rules).
- E.g. In state 2, "•" precedes the non-terminal E, so add all the rules that have E on the LHS, i.e. $E \to E + T$ and $E \to T$
- Now "•" also proceeds T, so add all the rules with T on the LHS as well, i.e. $T \to id$

# Building an LR(0) automaton

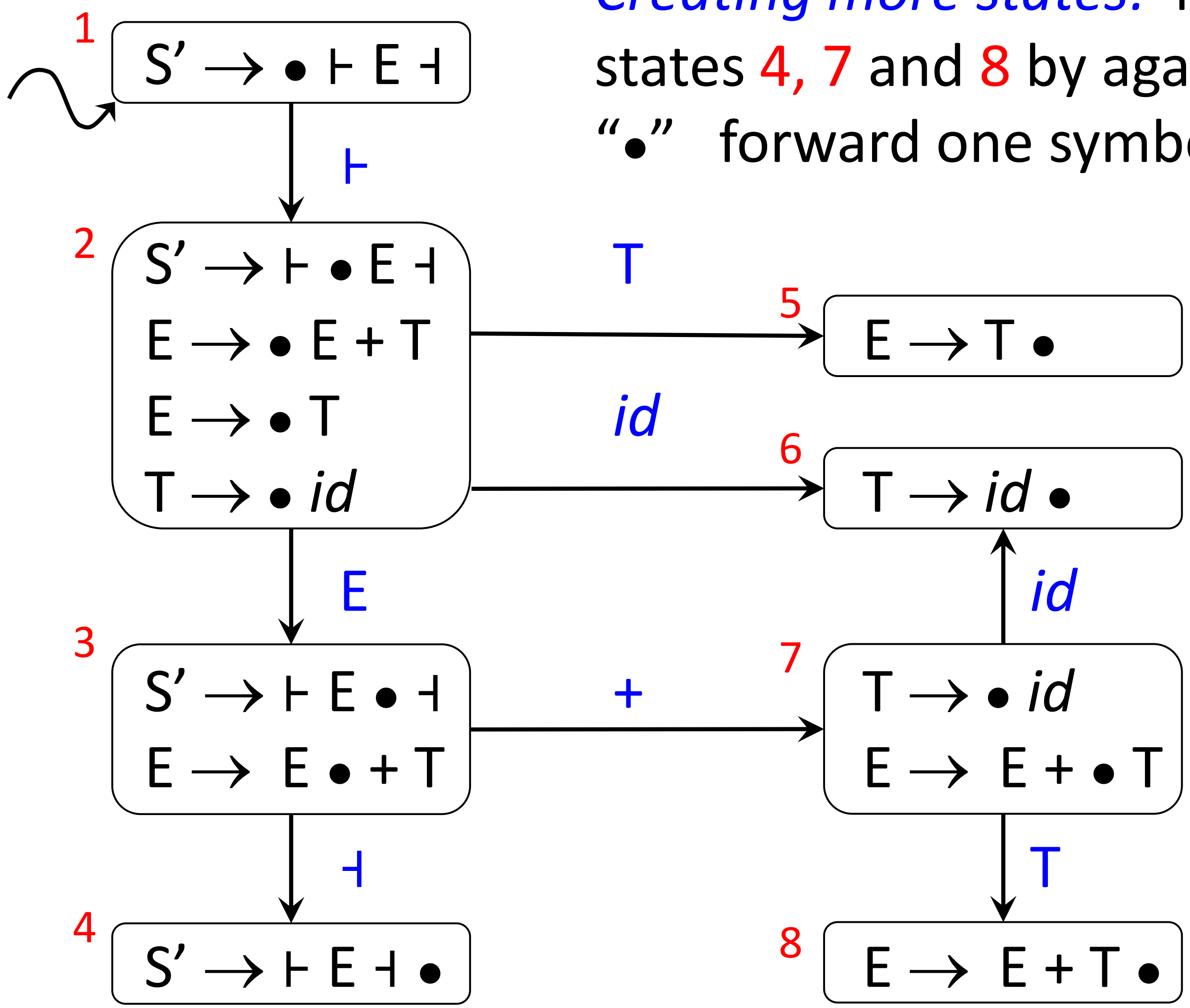



**Sample Grammar**

G: 1. S′ → ⊢ E ⊣
2. E → E + T
3. E → T
4. T → *id*

*Creating more states:* Since the "•" precedes E, T and *id* in state 2, there will be 3 transitions out of state 2, labelled E, T and *id*. In each new state, the "•" will move forward one symbol.

# Building an LR(0) automaton




*Creating more states:* next we add states 4, 7 and 8 by again moving the "•" forward one symbol.

# Using the Automaton

**The Algorithm**

**for each** input token
  **read** the stack (from the bottom up) + the current input
  **do** the action indicated for the current input
    **if** there's a transition out of current state with current input
     **then** shift (push) that input onto the stack
    **if** current state has only one item **and** the rightmost symbol is "•"
     **then** we know we can reduce i.e. {
        **pop** the RHS of the stack,
        **reread** the stack (from the bottom-up),
        **follow** the transition for the LHS,
        **push** the LHS onto the stack
     }
**if** S' is on the stack when all input is read
    **then** ACCEPT

E.g. states 4, 5, 6 and 8 on the previous slide

# Building an LR(0) automaton

**Sample Grammar**

G: 1. $S' \rightarrow \vdash E \dashv$
   2. $E \rightarrow E + T$
   3. $E \rightarrow T$
   4. $T \rightarrow id$

- *Task:* Use the grammar G and the automaton to parse the input $\vdash$ *id+id+id* $\dashv$
- Besides tracking the read and unread input, and the stack that tracks the symbols, also keep track of the states the automaton has been in, i.e. the States Stack

# Using the LR(0) Automaton

**Simulation**

| | Symbol Stack | States Stack | Input Read | Unread Input | Action |
|---|---|---|---|---|---|
| 1 | ε | 1 | ε | ⊢ *id*+*id*+*id* ⊣ | shift ⊢ |
| 2 | ⊢ | 1 2 | ⊢ | *id*+*id*+*id* ⊣ | shift *id* |
| 3 | ⊢ *id* | 1 2 6 | ⊢ *id* | +*id*+*id* ⊣ | |

1. Starting in state 1, read/shift ⊢,
   according to the automaton, move to state 2
   update the State Stack (now: 1 2)
2. State 2: read/shift *id*,
   according to the automaton, move to state 6

# Using the LR(0) Automaton

| | Symbol Stack | States Stack | Input Read | Unread Input | Action |
|---|---|---|---|---|---|
| 3 | ⊢ *id* | 1 2 6 | ⊢ *id* | +*id*+*id* ⊣ | reduce *id* |
| 4 | ⊢ T | 1 2 5 | ⊢ *id* | +*id*+*id* ⊣ | |

3. state 6 has only one item (T → *id* •) and "•" is the rightmost symbol, so reduce *id*
   - pop the RHS off of the symbol stack, i.e. *id*
   - reread the symbol stack (now: ⊢ ) and go corresponding state (2)
   - push the LHS of the rule you have just reduced (the rule was T → *id* •, so push T)
   - go to the appropriate state for T
   - from state 2 go to 5, updating the state stack (now: 1 2 5)

# Using the LR(0) Automaton

| | Symbol Stack | States Stack | Input Read | Unread Input | Action |
|---|---|---|---|---|---|
| 4 | ⊢ T | 1 2 5 | ⊢ *id* | *+id+id* ⊣ | reduce T |
| 5 | ⊢ E | 1 2 3 | ⊢ *id* | *+id+id* ⊣ | |

4. state 6 has only one item (E $\rightarrow$ T •) and "•" is the rightmost symbol, so reduce T:
   - pop the RHS off of the symbol stack, i.e. T
   - reread the symbol stack, i.e. ⊢, and go corresponding state (2)
   - push the LHS of the rule you have just reduced (the rule was E $\rightarrow$ T •, so push E)
   - go to the appropriate state for E, i.e. go from state 2 go to state 3 and update state stack (now 1 2 3)

# Using the LR(0) Automaton

| | Symbol Stack | States Stack | Input Read | Unread Input | Action |
|---|---|---|---|---|---|
| 5 | ⊢ E | 1 2 3 | ⊢ *id* | +*id*+*id* ⊣ | shift + |
| 6 | ⊢ E + | 1 2 3 7 | ⊢ *id*+ | *id*+*id* ⊣ | shift *id* |
| 7 | ⊢ E + *id* | 1 2 3 7 | ⊢ *id*+*id* | +*id* ⊣ | reduce *id* |

5. in state 3, read/shift + , move to state 7
6. in state 7, read/shift *id*, move to state 6
7. state 6 has only one item (T $\rightarrow$ *id* •) and "•" is the rightmost symbol, so reduce *id*:
   - pop the RHS off of the symbol stack, i.e. *id*
   - reread the symbol stack, i.e. ⊢ E +, and go corresponding state (1 2 3 7)

# Using the LR(0) Automaton

| | Symbol Stack | States Stack | Input Read | Unread Input | Action |
|---|---|---|---|---|---|
| 7 | ⊢ E + *id* | 1 2 3 7 | ⊢ *id*+*id* | +*id* ⊣ | reduce *id* |
| 8 | ⊢ E + T | 1 2 3 7 8 | ⊢ *id*+*id* | +*id* ⊣ | reduce E + T |

- push the LHS of the rule you have just reduced (the rule was T $\rightarrow$ *id* ●, so push T)
- go to the appropriate state for T, i.e. from state 7 go to state 8, and update state stack (now: 1 2 3 7 8)
- etc…

The next two slides illustrate the entire parsing of ⊢ *id*+*id*+*id* ⊣ …

# Using the LR(0) Automaton

| | Symbol Stack | States Stack | Input Read | Unread Input | Action |
|---|---|---|---|---|---|
| 1 | ε | 1 | ε | ⊢ *id*+*id*+*id* ⊣ | shift ⊢ |
| 2 | ⊢ | 1 2 | ⊢ | *id*+*id*+*id* ⊣ | shift *id* |
| 3 | ⊢ *id* | 1 2 6 | ⊢ *id* | +*id*+*id* ⊣ | reduce *id* |
| 4 | ⊢ T | 1 2 5 | ⊢ *id* | +*id*+*id* ⊣ | reduce T |
| 5 | ⊢ E | 1 2 3 | ⊢ *id* | +*id*+*id* ⊣ | shift + |
| 6 | ⊢ E + | 1 2 3 7 | ⊢ *id*+ | *id*+*id* ⊣ | shift *id* |
| 7 | ⊢ E + *id* | 1 2 3 7 6 | ⊢ *id*+*id* | +*id* ⊣ | reduce *id* |
| 8 | ⊢ E + T | 1 2 3 7 8 | ⊢ *id*+*id* | +*id* ⊣ | reduce E + T |
| 9 | ⊢ E | 1 2 3 | ⊢ *id*+*id* | +*id* ⊣ | shift + |
| 10 | ⊢ E + | 1 2 3 7 | ⊢ *id*+*id*+ | *id* ⊣ | shift *id* |

# Using the LR(0) Automaton

| | Symbol Stack | States Stack | Input Read | Unread Input | Action |
|---|---|---|---|---|---|
| 10 | ⊢ E + | 1 2 3 7 | ⊢ *id*+*id*+ | *id* ⊣ | shift *id* |
| 11 | ⊢ E + *id* | 1 2 3 7 6 | ⊢ *id*+*id*+*id* | ⊣ | reduce *id* |
| 12 | ⊢ E + T | 1 2 3 7 8 | ⊢ *id*+*id*+*id* | ⊣ | reduce E + T |
| 13 | ⊢ E | 1 2 3 | ⊢ *id*+*id*+*id* | ⊣ | shift ⊣ |
| 14 | ⊢ E ⊣ | 1 2 3 4 | ⊢ *id*+*id*+*id*⊣ | ε | reduce ⊢ E ⊣ |
| 15 | S’ | 1 | ⊢ *id*+*id*+*id*⊣ | ε | accept |

**Note**

- Line 10 has been repeated from the previous table.