# Bottom-up Parsing

Dolors Sala

(Original slides by Yannis Smaragdakis, Univ Athens, Lecture 5)

# Bottom-up Parsing

- More general than top-down parsing
  - And as efficient
  - Builds on the ideas in top-down parsing

- Specific algorithm: LR parsing
  - L means that (input) tokens are read left to right
  - R means that it constructs a rightmost derivation
  - Donald Knuth (1965), "*On the translation of languages from Left to Right*"

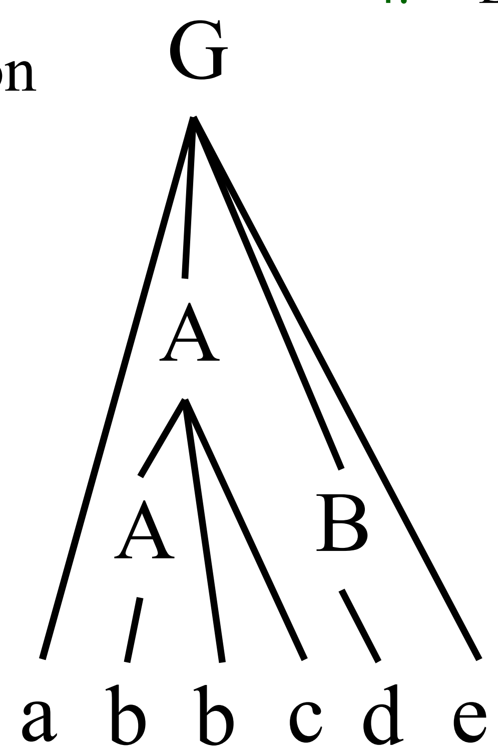
# A Bottom-up Example

- Start with input stream
  - Leaves of parse tree
- Build up towards goal symbol
  - Called reducing
  - Construct the reverse derivation

Production Rules

1. G → aABe
2. A → Abc
3. | b
4. B → d

| Rule | Sentential form | |
|---|---|---|
| - | abbcde | abbcde |
| 3 | aAbcde | aAbcde |
| 2 | aAde | aAde |
| 4 | aABe | aABe |
| 1 | G | |



# The Idea

- An LR parser reduces a string to the start symbol by inverting productions:

  str input string of terminals

  repeat

  - Identify $\beta$ in str such that $A \rightarrow \beta$ is a production

    (i.e., str = $\alpha\beta\gamma$)

  - Replace $\beta$ by A in str

    (i.e., str becomes $\alpha A\gamma$)

  until str = G

- LR parsers:
  - They can handle left-recursions
  - They don't need left factoring

# It seems Simple

- How to choose the correct reduction?
  - It is not as simple as find a reduction in the right-hand side and apply it
- Example: input abbcde

Production Rules

1. G → <u>a</u>AB<u>e</u>
2. A → A<u>b</u> <u>c</u>
3. | <u>b</u>
4. B→ <u>d</u>

| Rule | Sentential form |
|---|---|
| - | abbcde |
| 3 | aAbcde |
| 3 | aAAcde |
| ? | What reduction? |

- aAAcde is not part of any valid sentential form

# Key Concepts

- How do we make it work?
  - How do we know we do not get blocked?
  - How do we decide the next reduction?
  - How do we find it efficiently?
- Key
  - We are constructing the right-most derivation
  - Grammar is unambiguous
    - Unique right-most derivation for every string
    - Unique production applied at each forward step
    - Unique correct reduction at each backward step

# LR Parsing: Stack with 2 Operations

- State of the parser:

$$\alpha \mid \gamma$$

  - $\alpha$ is a stack of terminals and non-terminals
  - $\gamma$ is string of unexamined terminals
  - | the current input reading position

Production Rules

1. E → E + ( E )
2. | int

- Two operations
  - Shift: read next terminal, push on the stack

$$E + ( \mid int ) \rightarrow E + ( int \mid )$$

  - Reduce: pop RHS symbols off stack, push LHS

$$E + ( E + ( E ) \mid ) \rightarrow E + ( E \mid )$$

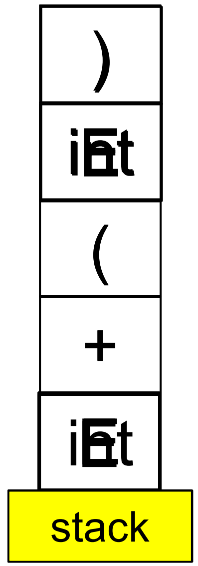
# Example

Production Rules

1. E → E + ( E )
2. | int

1. | int + ( int ) + ( int ) Nothing on stack, get next token
2. int | + ( int ) + ( int ) Shift: push int
3. int | + ( int ) + ( int ) Reduce: pop int, push E
4. int + | ( int ) + ( int ) Shift: push +
5. int + ( | int ) + ( int ) Shift: push (
6. int + ( int | ) + ( int ) Shift: push int
7. int + ( int | ) + ( int ) Reduce: pop int, push E
8. int + ( int ) | + ( int ) Shift: push )
9. int + ( int ) | + ( int ) Reduce: pop x5 E + ( E ), push E
10. int + ( int ) + | ( int ) Shift: push +
11. int + ( int ) + ( | int ) Shift: push (
12. int + ( int ) + ( int | ) Shift: push int
13. int + ( int ) + ( int | ) Reduce: pop int, push E
14. int + ( int ) + ( int ) | Shift: push )
15. int + ( int ) + ( int ) | Reduce: pop x5 E + ( E ), push E

Finished

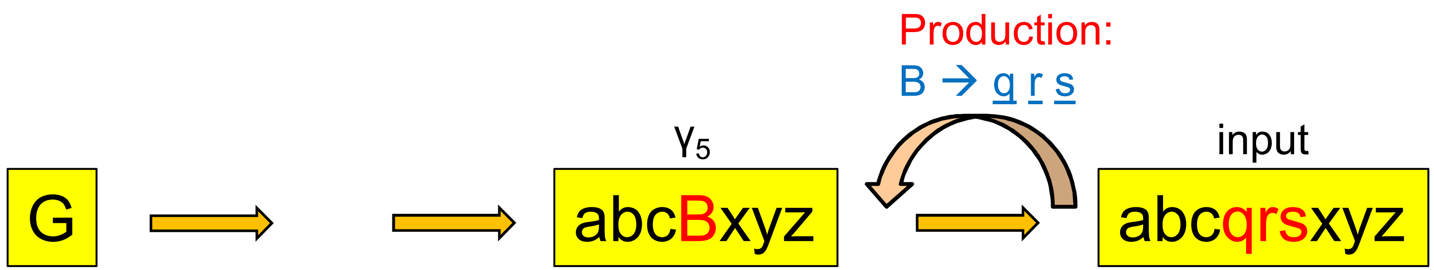


# Why does it work?

- Right-most derivation

$$G \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_3 \rightarrow \gamma_4 \rightarrow \gamma_5 \rightarrow \text{input}$$

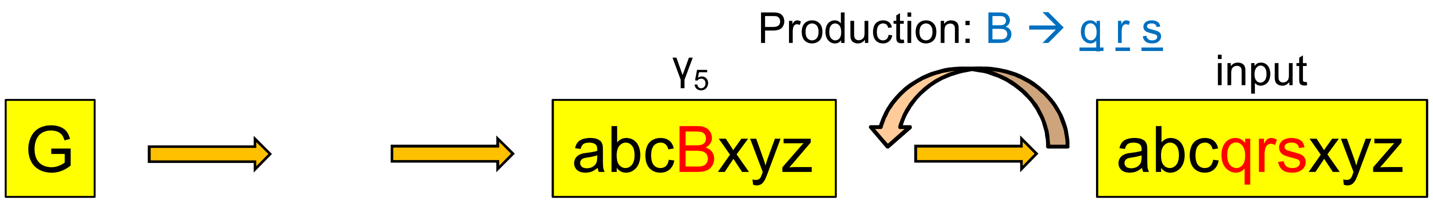
- Going backwards, start at the input and last step



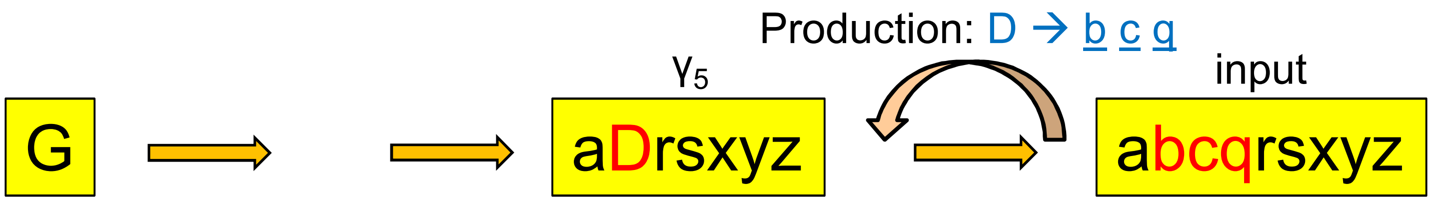

- To reverse this step:
  - Read input until rhs (q r s) are on top of the stack
  - Reduce rhs (q r s) to lhs (B)

# Right-most Derivation

- Could it be an alternative reduction?




- If it exists one such




- If it exists it means there are 2 right-most derivations for the same string
  - This would mean that the grammar is ambiguous
- Therefore, it cannot be an alternative reduction

# LR Parsing

**repeat**

**if** top symbols on stack match β for some A → β

Reduce: "found an A"

Pop those β symbols off

Push A on stack

**else** Get next token from scanner

**if** token is useful

Shift "still working on something"

Push on stack

**else** error - stop

**until** stack contains goal **and** no more input

# Key Problems

How do we know when to shift or reduce?

- Shifts
  - Default behavior: shift when there is no reduction
  - Still need to handle errors
- Reductions
  - At any given step, reduction is unique
  - Matching production occurs at top of stack
  - Problem: How to efficiently find the production to apply

# Identifying Reductions

Productions: $B \rightarrow \underline{q}\ \underline{r}\ \underline{s}$

$F \rightarrow \underline{c}\ B\ \underline{x}$

- Cases

$\gamma_4$: abFyz ← $\gamma_5$: abcBxyz ← abcqrsxyz

- Parsing state:
  - Input: $\underline{a}\ \underline{b}\ \underline{c}\ \underline{q}\ \underline{r}\ \underline{s}$ | $\underline{x}\ \underline{y}\ \underline{z}$
  - Stack: $\underline{a}\ \underline{b}\ \underline{c}$ B

- What is on the stack?
  - Sequence of terminals and non-terminals
  - All applicable reductions, except the last, already applied
  - Called a viable prefix

# Viable Prefixes: Properties

- Viable prefixes are a regular language
  - They can be implemented with an automata

- Automata to identify viable prefixes
  - Input: stack contents (mix terminals & non-terminals)
  - Each state represents either
    - A right sentential form labeled with the reduction to apply
    - A viable prefix labeled with tokens to expect next

# Shift/Reduce DFA

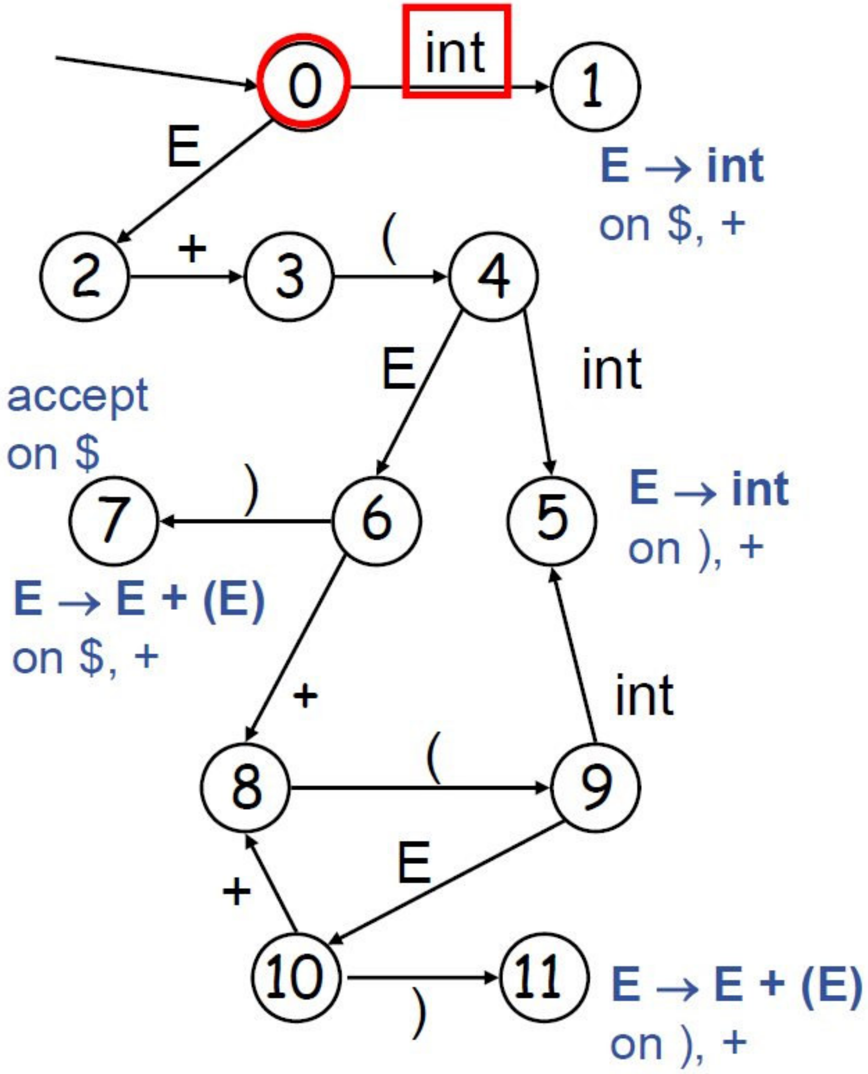
- Using the DFA
  - At each parsing step run the DFA on stack contents
  - Examine the resulting state X and the token t immediately following | in the input stream
    - If X has an outgoing edge labeled t then shift
    - If X is labeled "A → β on t" then reduce

- Example:

Production Rules

1. E → E + ( E )
2. | int

# Shift/Reduce DFA: Example (1)



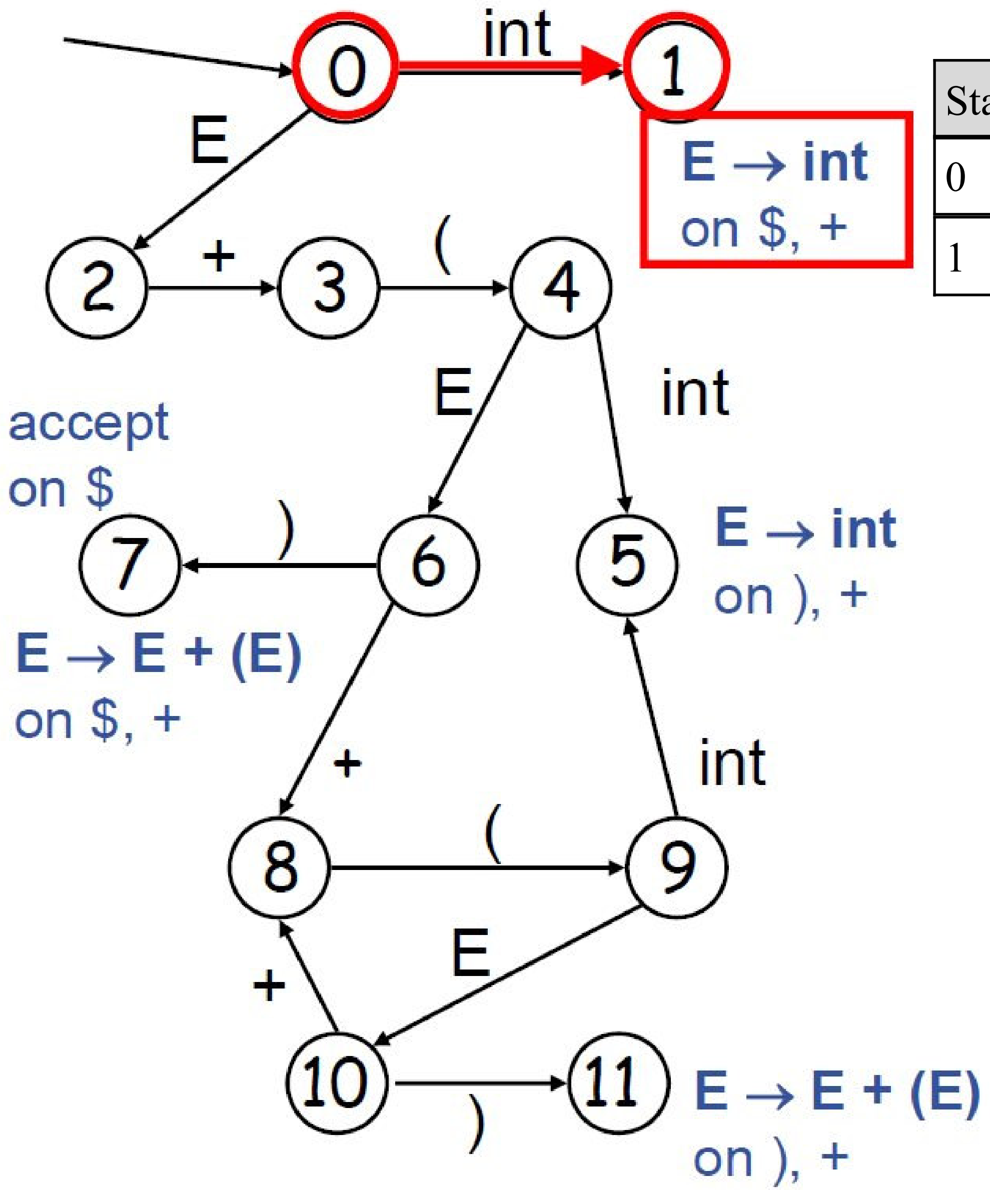

| State | Input | Stack | Operations |
|---|---|---|---|
| 0 | \| int + (int) + (int)$ | | |

Production Rules

1. E → E + ( E )
2. | int

(Later we see how to derive he automata)

# Shift/Reduce DFA: Example (2)

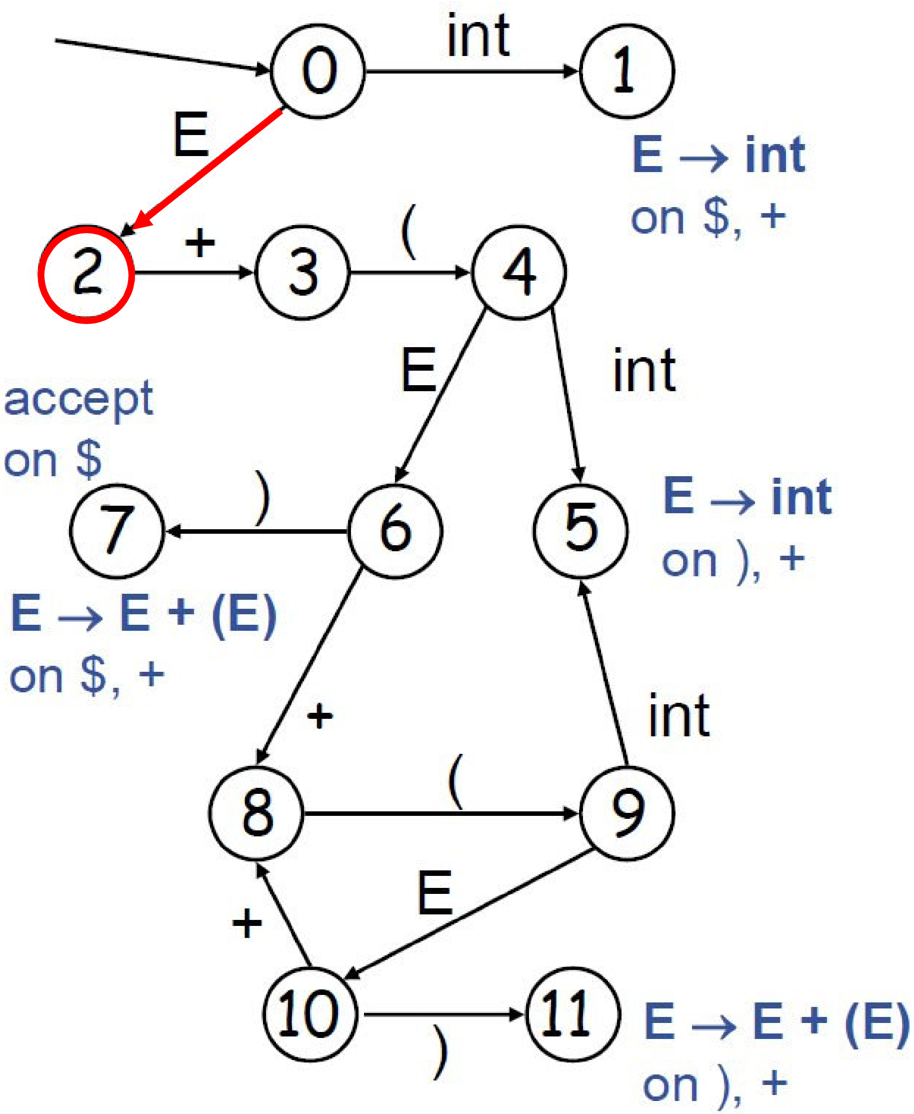



| State | Input | Stack | Operations |
|---|---|---|---|
| 0 | \| int + (int) + (int)$ | | |
| 1 | int \| + (int) + (int)$ | int | shift |

Production Rules

1. E → E + ( E )
2. | int

# Shift/Reduce DFA: Example (3)

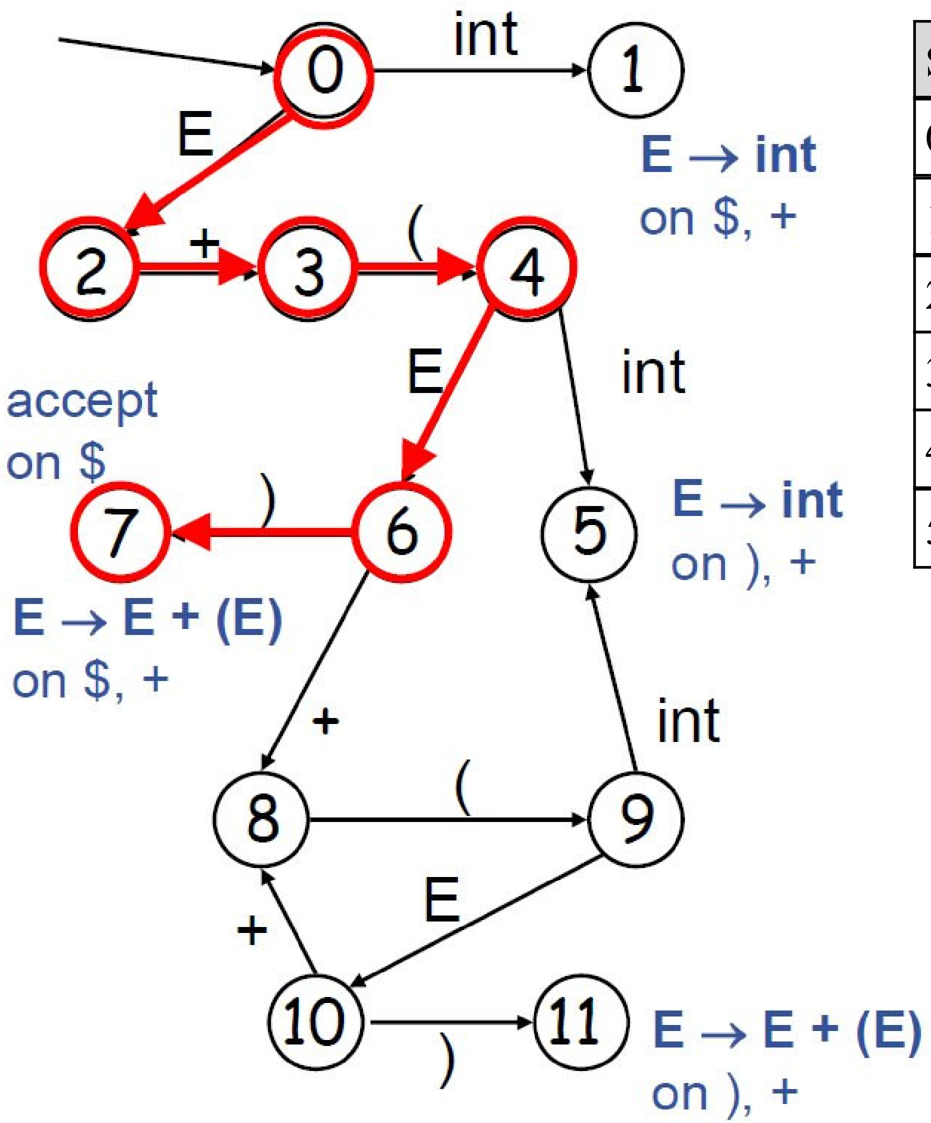



| State | Input | Stack | Operations |
|---|---|---|---|
| 0 | \| int + (int) + (int)$ | | |
| 1 | int \| + (int) + (int)$ | int | shift |
| 2 | int \| + (int) + (int)$ | E | reduce R2 |

Production Rules

1. E → E + ( E )
2. | int

# Shift/Reduce DFA: Example (4)



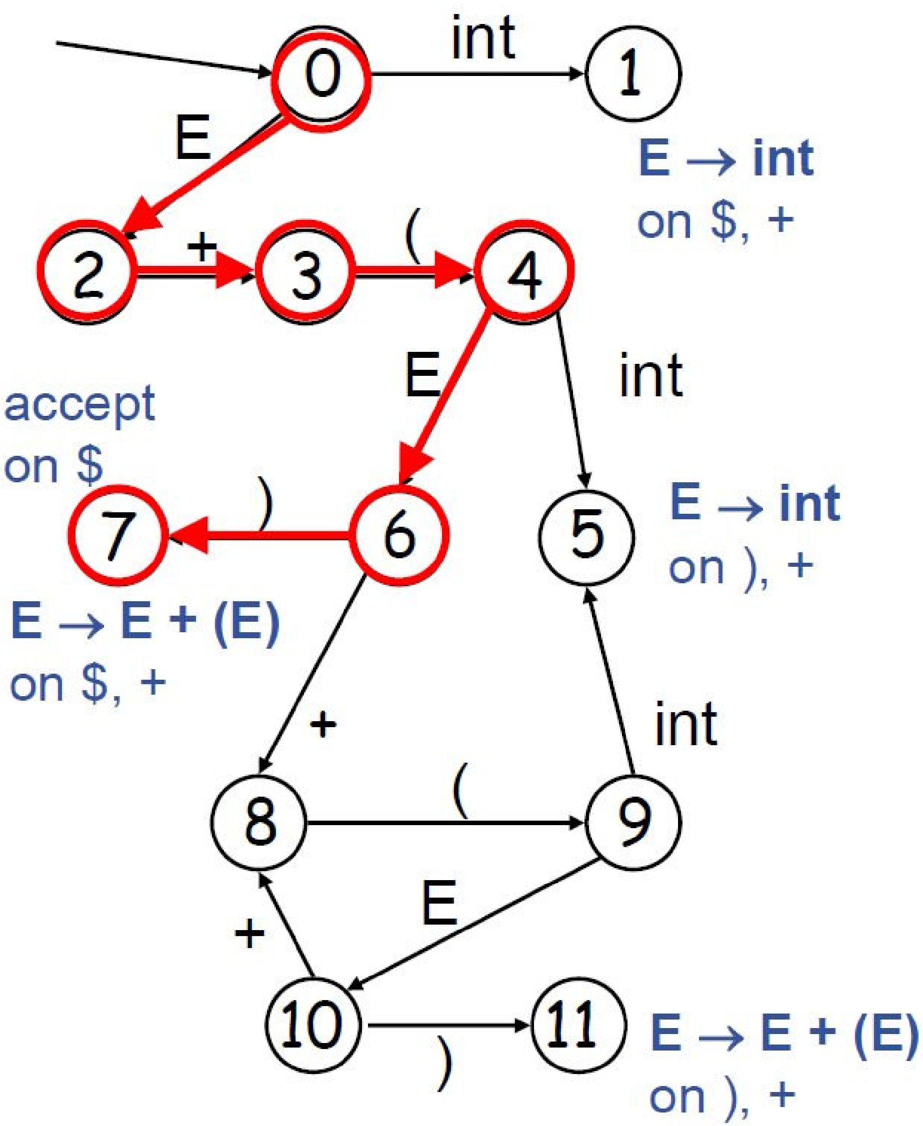

| State | Input | Stack | Operations |
|---|---|---|---|
| 0 | \| int + (int) + (int)$ | | |
| 1 | int \| + (int) + (int)$ | int | shift |
| 2 | int \| + (int) + (int)$ | E | reduce R2 |
| 3 | int + \| (int) + (int)$ | E + | shift |
| 4 | int + ( \| int) + (int)$ | E + ( | shift |
| 5 | int + (int \| ) + (int)$ | E + ( int | shift |

Production Rules

1. E → E + ( E )
2. | int

# Shift/Reduce DFA: Example (4)

ENGINYERIA UPF

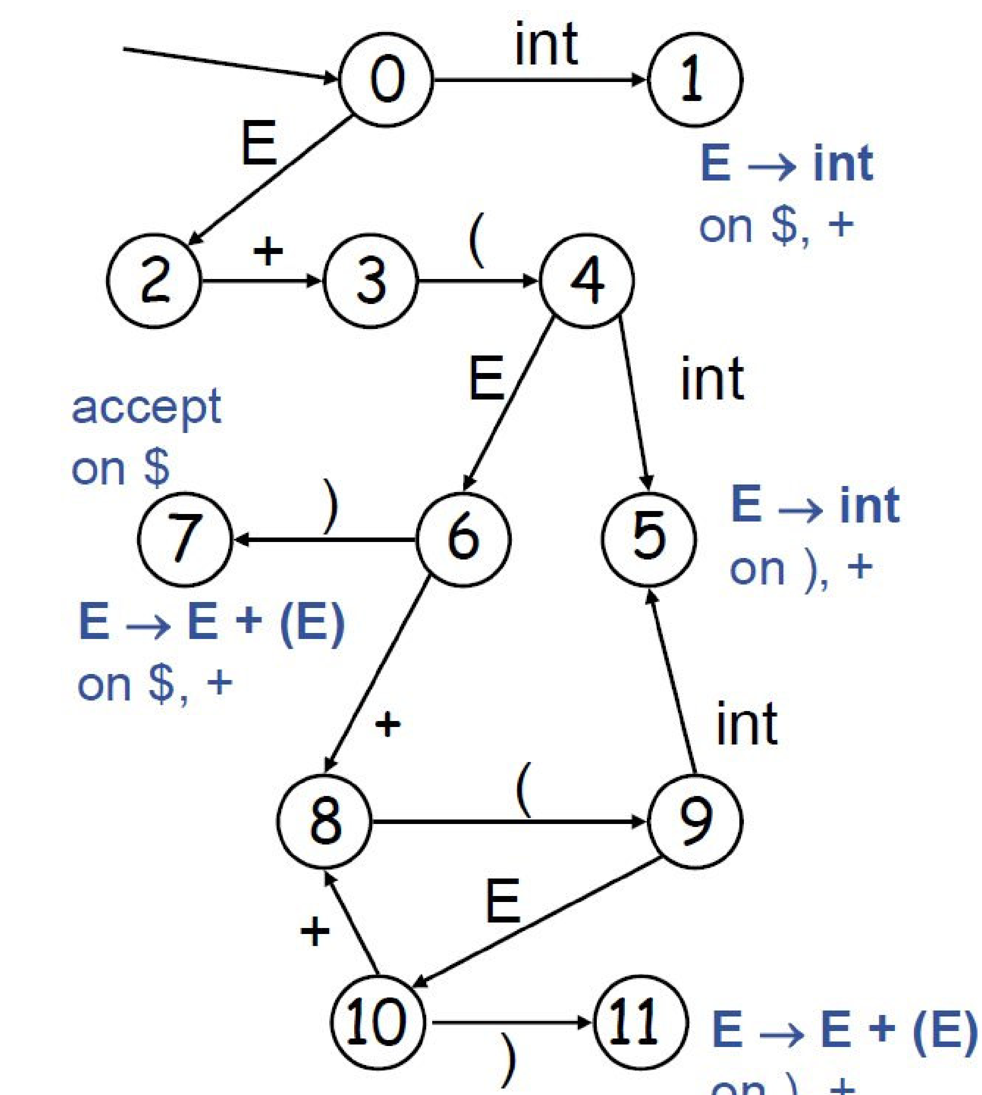



| State | Input | Stack | Operations |
|---|---|---|---|
| 0 | \| int + (int) + (int)$ | | |
| 1 | int \| + (int) + (int)$ | int | shift |
| 2 | int \| + (int) + (int)$ | E | reduce R2 |
| 3 | int + \| (int) + (int)$ | E + | shift |
| 4 | int + ( \| int) + (int)$ | E + ( | shift |
| 5 | int + (int \| ) + (int)$ | E + ( int | shift |
| 6 | int + (int \| ) + (int)$ | E + ( E | R2 |
| 7 | int + (int) \| + (int)$ | E + ( E ) | shift |
| 2 | int + (int) \| + (int)$ | E | R1 |

Production Rules

1. E → E + ( E )
2. | int

# Shift/Reduce DFA: Example (4)



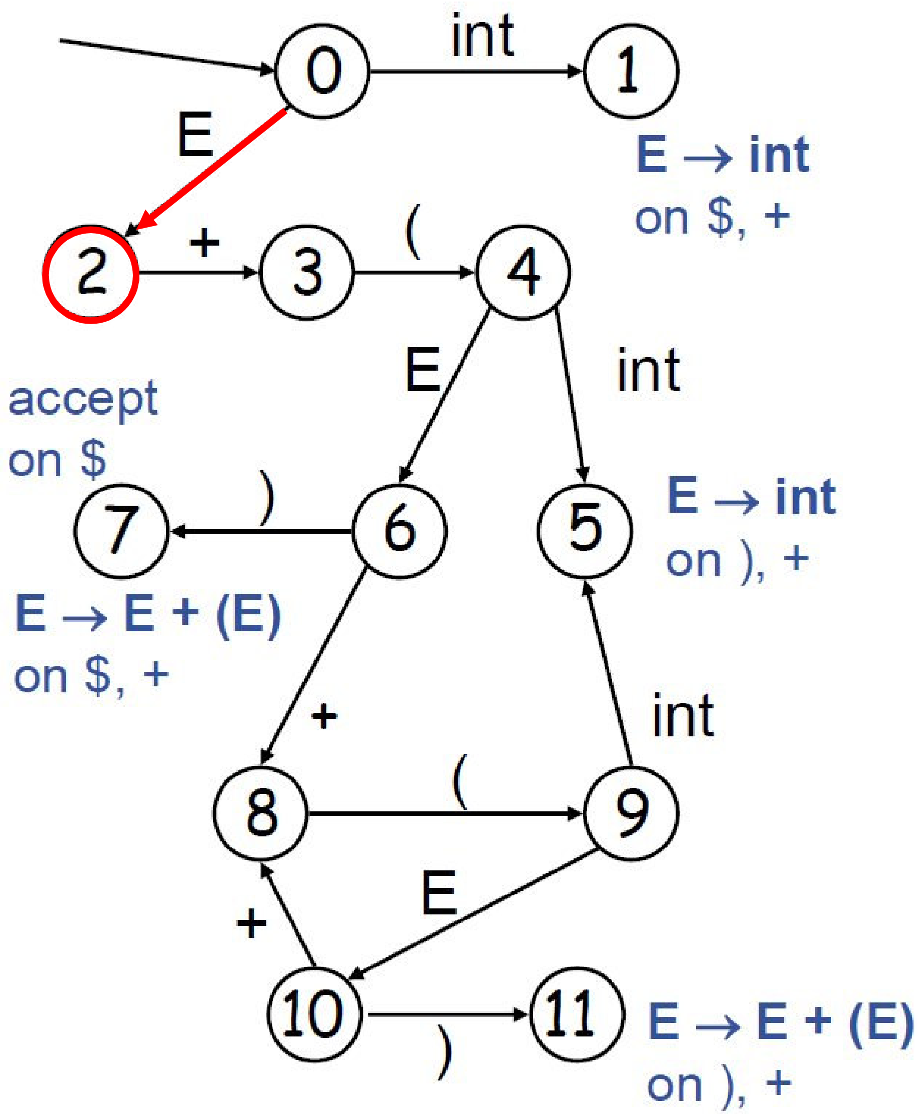

| State | Input | Stack | Operations |
|---|---|---|---|
| 0 | \| int + (int) + (int)$ | | |
| 1 | int \| + (int) + (int)$ | int | shift |
| 2 | int \| + (int) + (int)$ | E | reduce R2 |
| 3 | int + \| (int) + (int)$ | E + | shift |
| 4 | int + ( \| int) + (int)$ | E + ( | shift |
| 5 | int + (int \| ) + (int)$ | E + ( int | shift |
| 6 | int + (int \| ) + (int)$ | E + ( E | R2 |
| 7 | int + (int) \| + (int)$ | E + ( E ) | shift |
| 2 | int + (int) \| + (int)$ | E | R1 |
| 3 | int + (int) +\| (int)$ | E + | shift |
| 4 | int + (int) + ( \| int)$ | E + ( | shift |
| 5 | int + (int) + (int \| )$ | E + ( int | shift |
| 6 | int + (int) + (int \| )$ | E + ( E | R2 |
| 7 | int + (int) + (int ) \| $ | E + ( E ) | shift |
| 2 | int + (int) + (int ) \| $ | E | R1 / accept |

Product.Rules

1 E → E + ( E )

2 | int

# Shift/Reduce DFA: Merge shift - reduce




| State | Input | Stack | Operations |
|---|---|---|---|
| 0 | \| int + (int) + (int)$ | | |
| 1 | int \| + (int) + (int)$ | int | shift |
| 2 | int \| + (int) + (int)$ | E | reduce R2 |

| | | | |
|---|---|---|---|
| 2 | int \| + (int) + (int)$ | E | shift/R2 |

Production Rules

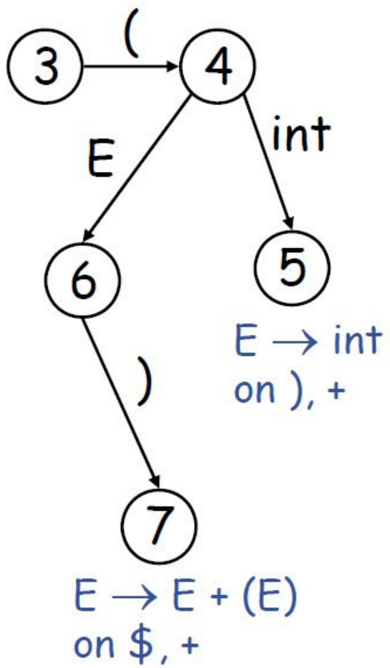
1. E → E + ( E )
2. | int

# Operations

- Each DFA state represents stack contents
  - At each step, we run the DFA to compute the new state
  - Two actions:
    - Shift: Push a new token
    - Reduce: Pop some symbols off, push a new symbol
- The DFA state can be stored in the stack
  - For each symbol on the stack, remember the DFA state that represents the contents up to the that point
  - Push a new token = go forward in the DFA
  - Pop a sequence of symbols = “unwind” to previous state in the DFA (stored in the stack)

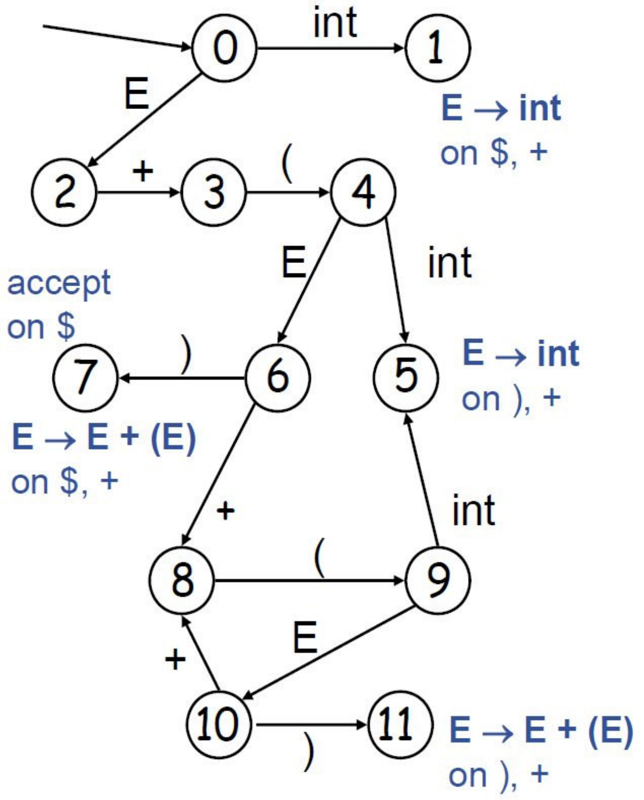
# Representing the shift / reduce DFA

- Combined table: not only the next state, but also the stack operations
  - Columns of table:
    - action for input symbols (terminal symbols)
    - goto for stack information (non-terminal symbols, applies reduction)




| | *action(state, token)* | | | | | *goto* |
|---|---|---|---|---|---|---|
| | int | + | ( | ) | $ | E |
| … | | | | | | |
| 3 | | | s4 | | | |
| 4 | s5 | | | | | g6 |
| 5 | | $r_{E\text{-->}int}$ | | $r_{E\text{-->}int}$ | | |
| 6 | | s8 | | s7 | | |
| 7 | | $r_{E\text{-->}E+(E)}$ | | | $r_{E\text{-->}E+(E)}$ | |
| … | | | | | | |

| | action(state, token) | | | | | goto |
|---|---|---|---|---|---|---|
| State | int | + | ( | ) | $ | E |
| 0 | 1 | | | | | g2 |
| 1 | | $r_{E\rightarrow int}$ 0 | | | $r_{E\rightarrow int}$ 0 | |
| 2 | | 3 | | | accept | |
| 3 | | | 4 | | | |
| 4 | 5 | | | | | g6 |
| 5 | | $r_{E\rightarrow int}$ 4 | | $r_{E\rightarrow int}$ 4 | | |
| 6 | | 8 | | 7 | | |
| 7 | | $r_{E\rightarrow E+(E)}$ 0 | | | $r_{E\rightarrow E+(E)}$ 0 | |
| 8 | | | 9 | | | |
| 9 | 5 | | | | | g10 |
| 10 | | 8 | | 11 | | |
| 11 | | $r_{E\rightarrow E+(E)}$ 4 | | $r_{E\rightarrow E+(E)}$ 4 | | |




Production Rules

1 E → E + ( E )

2 | int

# How is the DFA constructed?

- What is on the stack?
  - Viable prefix – A piece of a sentential form

E + (
E + ( int
E + ( E + (

- Idea: we are part-way through some production

- Problem: productions can share pieces

- DFA state represent the set of candidate productions
  - Represents all the productions we could be working on
  - Notation: LR(1) item shows where we are and what we need to see

# Definition: LR Items

- An LR(1) item is a pair:

  $[A \rightarrow \alpha \bullet \beta, \underline{a}]$

- $A \rightarrow \alpha\beta$ is a production
- $\underline{a}$ is a terminal (the lookahead terminal)
- LR(1) means 1 lookahead terminal

- $\boxed{[A \rightarrow \alpha \bullet \beta, \underline{a}]}$ describes a context of the parser
  - We are trying to find an A followed by an $\underline{a}$, and
  - We have seen an $\alpha$
  - We need to see a string derived from $\beta\ \underline{a}$

# LR Items

- In context containing (position in the middle of rule)

  [E → E + • ( E ), +]

  – If "(" is next then we can shift to context containing

  [E → E + ( • E ), +]

- In the context containing (position at the end of rule)

  [E → E + (E ) •, +]

  – We can reduce with the E → E + ( E )
  – But only if a "+" follows

# LR Items

- Consider the item

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

  we expect a string derived from $E\,)\,+$

  There are 2 productions for $E$

  $E \rightarrow int$ and $E \rightarrow E + (E)$

- We extend the context with two more items:

  $E \rightarrow \bullet\, int, \,)$

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

- Each DFA state:
  - The set of items that represent all possible productions we could be working on – called the closure of the set of items

# Closure Example

- Starting context = **closure**({$S \rightarrow \bullet E, \$$})

1. $S \rightarrow \bullet E, \$$
2. $E \rightarrow \bullet E+(E), \$$
3. $E \rightarrow \bullet \text{int}, \$$
4. $E \rightarrow \bullet E+(E), +$
5. $E \rightarrow \bullet \text{int}, +$

- Abbreviated

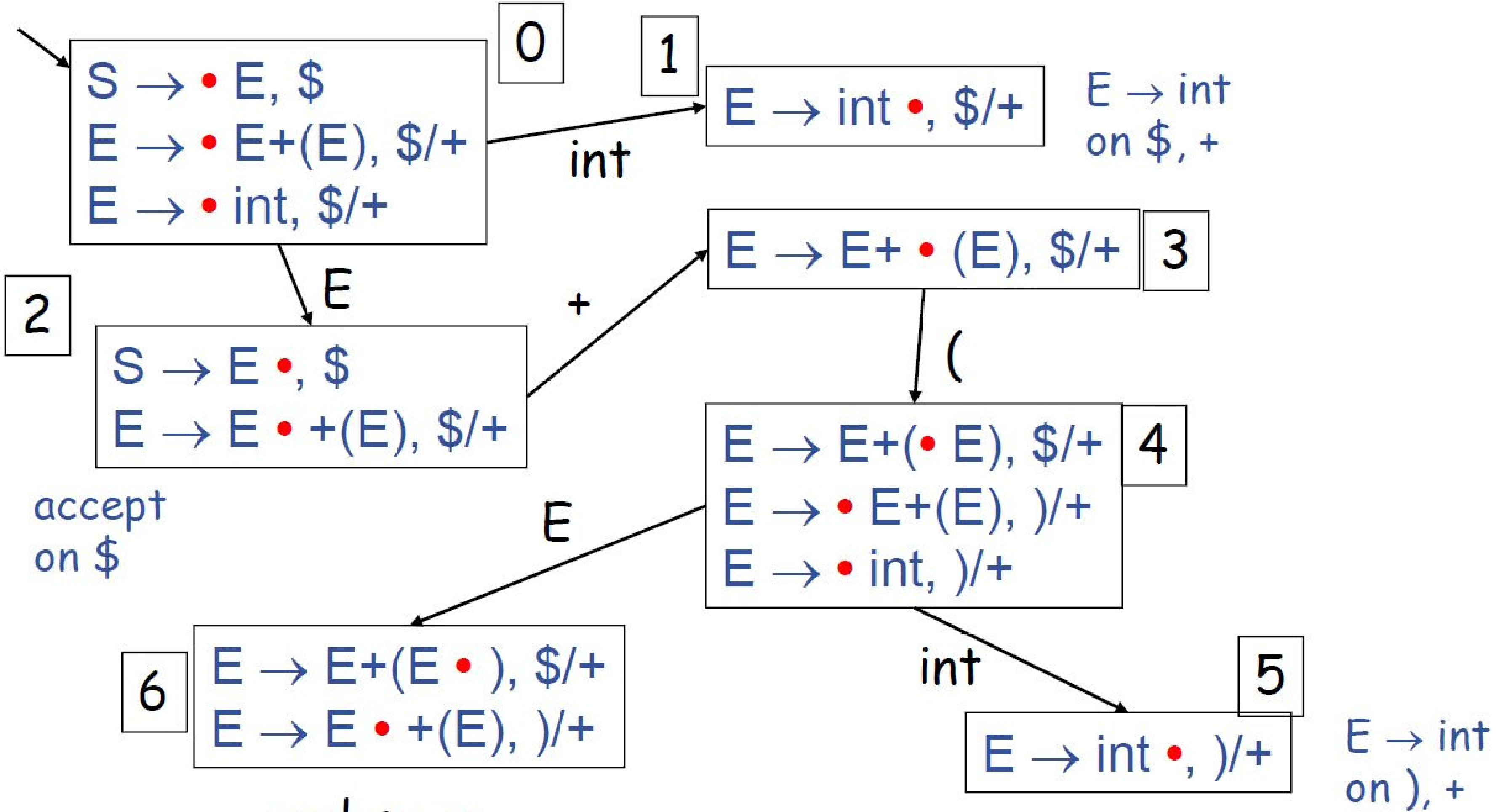
1. $S \rightarrow \bullet E, \$$
2. $E \rightarrow \bullet E+(E), \$/+$
3. $E \rightarrow \bullet \text{int}, \$/+$

# Example

Production Rules

1. $E \rightarrow \underline{E + ( E )}$
2. $\quad\quad | \ \underline{int}$

1. $S \rightarrow \bullet\ E, \$$
2. $E \rightarrow \bullet\ E+(E), \$/+$
3. $E \rightarrow \bullet\ int, \$/+$

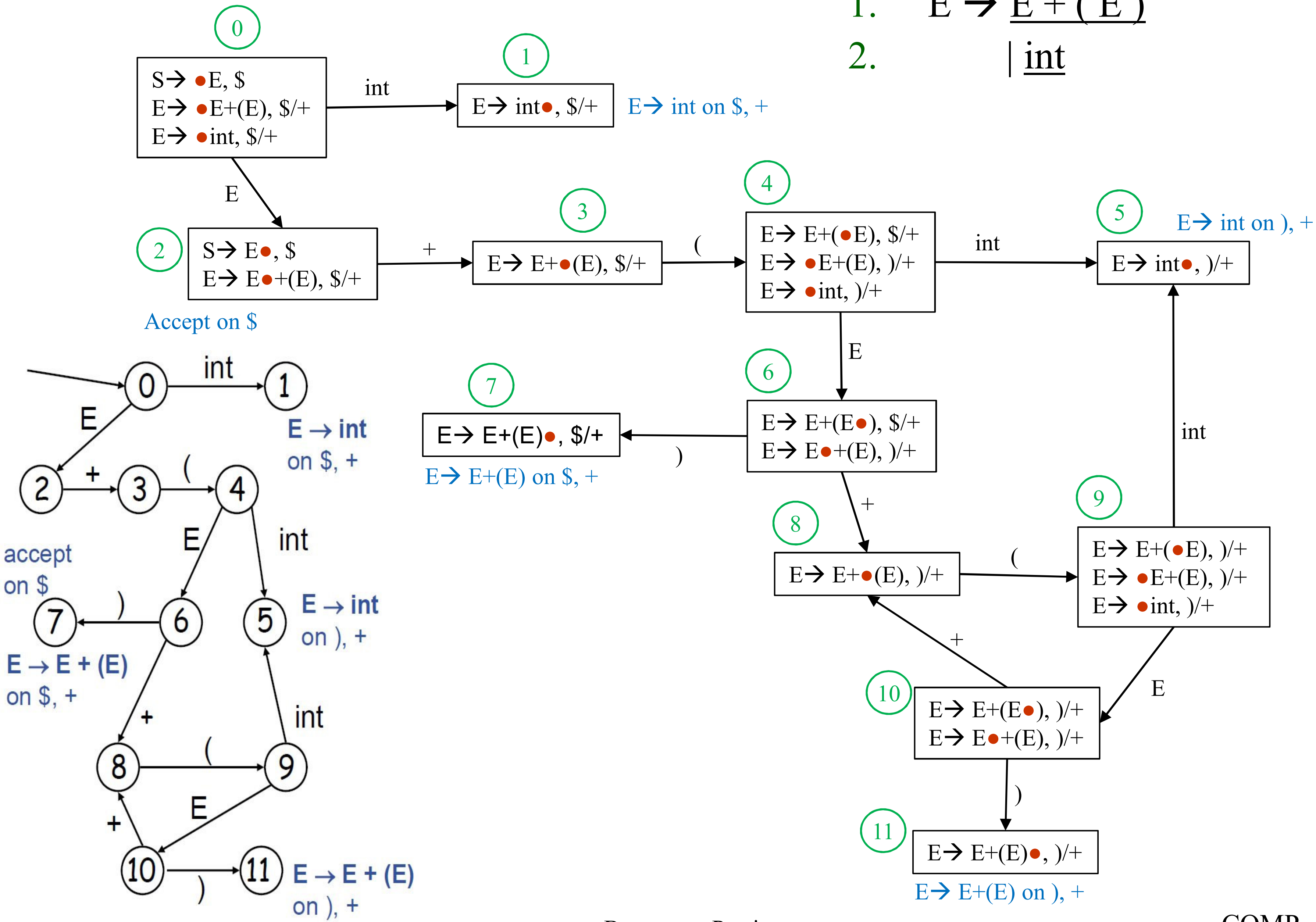



and so on…

# Example (Full DFA)

Production Rules

1. E → E + ( E )
2. | int



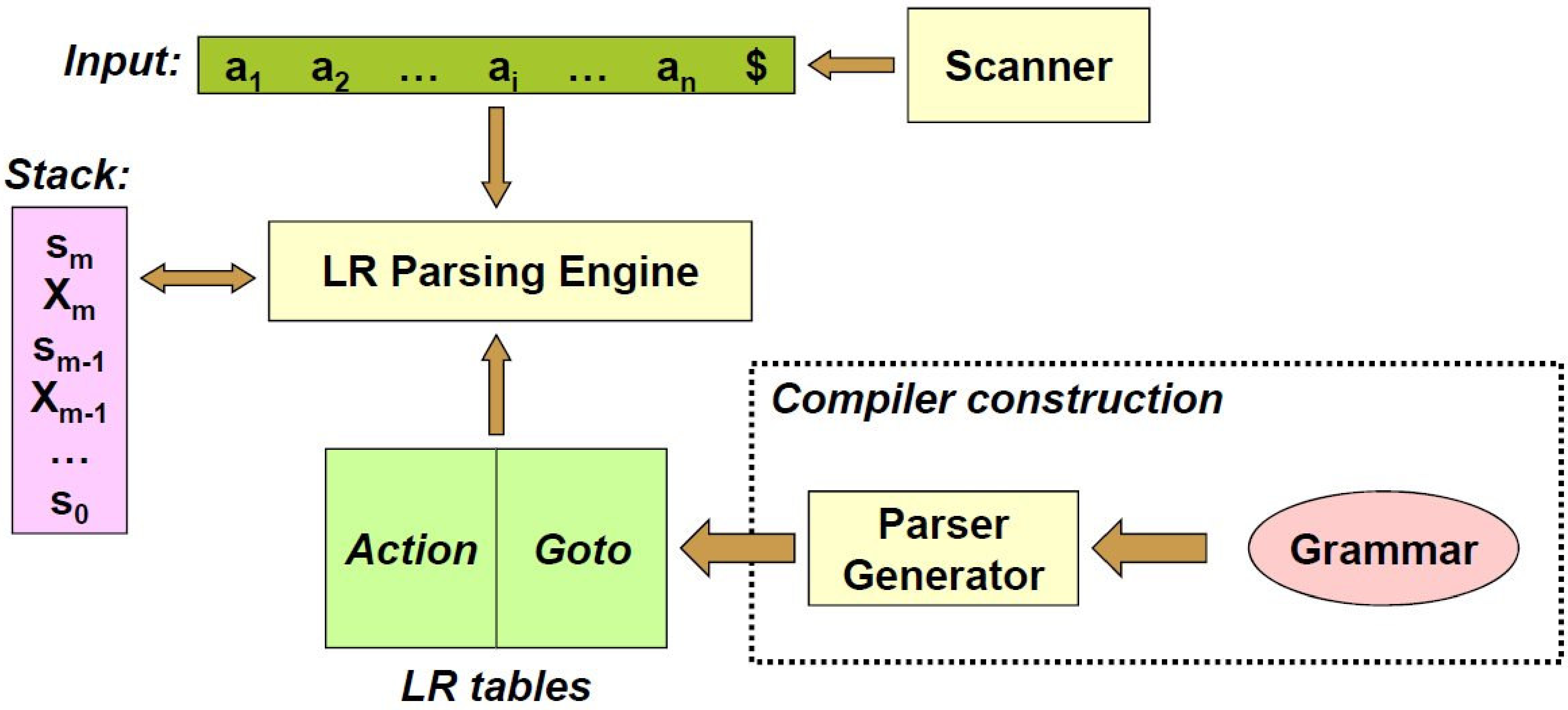
# Closure Operation

- Observation
  - At $A \rightarrow \alpha \bullet B\beta$ we expect to see $B\beta$ next
  - Means if $B \rightarrow \gamma$ is a production, then we could see a $\gamma$
- Algorithm

1. closure(Items) =
2. repeat
3. for each $[A \rightarrow +\alpha \bullet B\beta , \underline{a}]$ in Items
4. for each production $B \rightarrow \gamma$
5. for each $\underline{b} \in \text{FIRST}(\beta\underline{a})$
6. add $[B \rightarrow \bullet \gamma, \underline{b}]$ to Items
7. until Items is unchanged

| | action(state, token) | | | | | goto |
|---|---|---|---|---|---|---|
| State | int | + | ( | ) | $ | E |
| 0 | 1 | | | | | g2 |
| 1 | | $r_{E \rightarrow int}$ 0 | | | $r_{E \rightarrow int}$ 0 | |
| 2 | | 3 | | | accept | |
| 3 | | | 4 | | | |
| 4 | 5 | | | | | g6 |
| 5 | | $r_{E \rightarrow int}$ 4 | | $r_{E \rightarrow int}$ 4 | | |
| 6 | | 8 | | 7 | | |
| 7 | | $r_{E \rightarrow E+(E)}$ 0 | | | $r_{E \rightarrow E+(E)}$ 0 | |
| 8 | | | 9 | | | |
| 9 | 5 | | | | | g10 |
| 10 | | 8 | | 11 | | |
| 11 | | $r_{E \rightarrow E+(E)}$ 4 | | $r_{E \rightarrow E+(E)}$ 4 | | |

# LR Parsing



Input:
$a_1$ $a_2$ … $a_i$ … $a_n$ $
Scanner
Stack:
$s_m$
$X_m$
$s_{m-1}$
$X_{m-1}$
…
$s_0$
LR Parsing Engine
Compiler construction
Action
Goto
Parser Generator
Grammar
LR tables

# Issues with LR parsers

- What happens if a state contains

  $[A \rightarrow \alpha \bullet a\beta, \underline{b}]$ and $[Y \rightarrow \gamma \bullet, \underline{a}]$

- Then on input "a" we could either
  - Shift into state $[A \rightarrow \alpha \bullet a\beta, \underline{b}]$ or
  - Reduce with $Y \rightarrow \gamma$
- This is called a shift-reduce conflict
  - Typically due to ambiguity

- There are many more issues to consider
- Not covered in this course

# Summary of Parsing (Bottom-up)

- A more powerful parser: LR(1) (bottom-up parser)
- It starts from the input and replaces right-hand for the left-hand side of the production rules (bottom-up)
- It supports left-recursion
- It uses a DFA that defines the next state and action table and keeps track of the advance with an stack
- Shift-reduced automata Actions:
  - Reduce: match of production rule, pop all rhs symbols and push lhs
  - Shift: advance input and push symbol to stack
  - Accept: when original symbol and input all processed
  - Error: otherwise
- Items define the context of the parser
- Closure of a set of items define the possible production rules to apply