南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# Chapter 2: Context-Free Grammars & Syntax Analysis

Yepang Liu

liuyp1@sustech.edu.cn

# Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
- Top-Down Parsing Techniques
- Bottom-Up Parsing Techniques

# Bottom-Up Parsing

- **Problem definition:** Constructing a parse tree for an input string beginning at the leaves (terminals) and working up towards the root (start symbol of the grammar)

- It can be seen as a process of "reducing" a string $\omega$ to the start symbol of the grammar (a reverse process of derivation)

# Shift-Reduce Parsing

- Shift-reduce parsing (移入-归约分析) is a general style of bottom-up parsing:
  - A **stack** holds grammar symbols
  - An **input buffer** holds the rest of the string to be parsed
  - Two basic actions:
    - **Shift:** Move an input symbol (terminal) onto the stack
    - **Reduce:** Replace a string at the stack top with a non-terminal that can produce the string (the reverse of a rewrite step in a derivation)

# Shift-Reduce Parsing - Overview

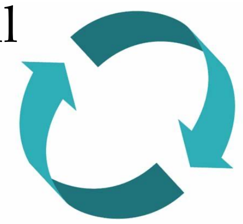
**Initial status:**

| STACK | INPUT |
|---|---|
| \$ | $\omega$\$ |

| **Actions:** |
|---|
| Shift |
| Reduce |
| Accept |
| Error |

**Shift-reduce process:**

- The parser **shifts** zero or more input symbols onto the stack, until it is ready to reduce a string $\beta$ on top of the stack
- **Reduce** $\beta$ to the head of the appropriate production



**The parser repeats the above cycle** until it has detected an error or the stack contains the start symbol and input is empty

# Shift-Reduce Parsing Example

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

| STACK | INPUT | ACTION |
|---|---|---|
| $ | $\mathbf{id}_1 * \mathbf{id}_2$ $ | shift |

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow ( E ) \mid \mathbf{id}$$

$\mathbf{id} * \mathbf{id}$

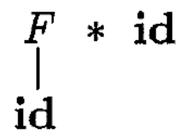
Initially, the tree only contains leaf nodes

# Shift-Reduce Parsing Example

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow ( E ) \mid \mathbf{id}$$

| STACK | INPUT | ACTION |
|---|---|---|
| $\$$ | $\mathbf{id}_1 * \mathbf{id}_2 \$$ | shift |
| $\$\,\mathbf{id}_1$ | $* \mathbf{id}_2 \$$ | reduce by $F \rightarrow \mathbf{id}$ |



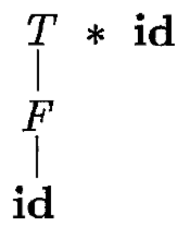

Tree "grows" when reduction happens

# Shift-Reduce Parsing Example

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

| STACK | INPUT | ACTION |
|---|---|---|
| $\$$ | $\mathbf{id}_1 * \mathbf{id}_2\ \$$ | shift |
| $\$\ \mathbf{id}_1$ | $* \mathbf{id}_2\ \$$ | reduce by $F \rightarrow \mathbf{id}$ |
| $\$\ F$ | $* \mathbf{id}_2\ \$$ | reduce by $T \rightarrow F$ |

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow ( E ) \mid \mathbf{id}$$



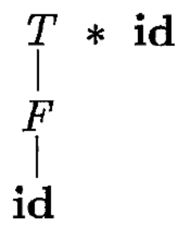

Tree "grows" when reduction happens

# Shift-Reduce Parsing Example

Parsing steps on input **id**$_1$ $*$ **id**$_2$

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow ( E ) \mid \textbf{id}$$

| STACK | INPUT | ACTION |
|---|---|---|
| $\$$ | $\textbf{id}_1 * \textbf{id}_2 \,\$$ | shift |
| $\$\, \textbf{id}_1$ | $* \textbf{id}_2 \,\$$ | reduce by $F \rightarrow \textbf{id}$ |
| $\$\, F$ | $* \textbf{id}_2 \,\$$ | reduce by $T \rightarrow F$ |
| $\$\, T$ | $* \textbf{id}_2 \,\$$ | shift |



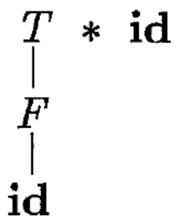

Tree does not change when shift happens

# Shift-Reduce Parsing Example

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow ( E ) \mid \mathbf{id}$$

| STACK | INPUT | ACTION |
|---|---|---|
| $\$$ | $\mathbf{id}_1 * \mathbf{id}_2 \,\$$ | shift |
| $\$\,\mathbf{id}_1$ | $* \mathbf{id}_2 \,\$$ | reduce by $F \rightarrow \mathbf{id}$ |
| $\$\,F$ | $* \mathbf{id}_2 \,\$$ | reduce by $T \rightarrow F$ |
| $\$\,T$ | $* \mathbf{id}_2 \,\$$ | shift |
| $\$\,T *$ | $\mathbf{id}_2 \,\$$ | shift |



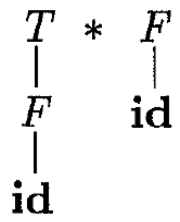

Tree does not change when shift happens

# Shift-Reduce Parsing Example

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to ( E ) \mid \mathbf{id}$$

| STACK | INPUT | ACTION |
|---|---|---|
| $\$$ | $\mathbf{id}_1 * \mathbf{id}_2 \$$ | shift |
| $\$\, \mathbf{id}_1$ | $* \mathbf{id}_2 \$$ | reduce by $F \to \mathbf{id}$ |
| $\$\, F$ | $* \mathbf{id}_2 \$$ | reduce by $T \to F$ |
| $\$\, T$ | $* \mathbf{id}_2 \$$ | shift |
| $\$\, T *$ | $\mathbf{id}_2 \$$ | shift |
| $\$\, T * \mathbf{id}_2$ | $\$$ | reduce by $F \to \mathbf{id}$ |



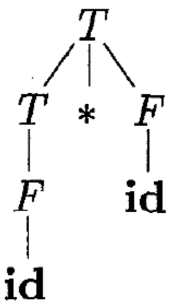

Tree "grows" when reduction happens

# Shift-Reduce Parsing Example

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow ( E ) \mid \mathbf{id}$$

| STACK | INPUT | ACTION |
|---|---|---|
| $\$$ | $\mathbf{id}_1 * \mathbf{id}_2 \$$ | shift |
| $\$ \mathbf{id}_1$ | $* \mathbf{id}_2 \$$ | reduce by $F \rightarrow \mathbf{id}$ |
| $\$ F$ | $* \mathbf{id}_2 \$$ | reduce by $T \rightarrow F$ |
| $\$ T$ | $* \mathbf{id}_2 \$$ | shift |
| $\$ T *$ | $\mathbf{id}_2 \$$ | shift |
| $\$ T * \mathbf{id}_2$ | $\$$ | reduce by $F \rightarrow \mathbf{id}$ |
| $\$ T * F$ | $\$$ | reduce by $T \rightarrow T * F$ |



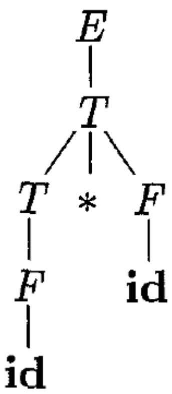

Tree "grows" when reduction happens

# Shift-Reduce Parsing Example

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow ( E ) \mid \mathbf{id}$$

| STACK | INPUT | ACTION |
|---|---|---|
| $\$$ | $\mathbf{id}_1 * \mathbf{id}_2 \$$ | shift |
| $\$ \mathbf{id}_1$ | $* \mathbf{id}_2 \$$ | reduce by $F \rightarrow \mathbf{id}$ |
| $\$ F$ | $* \mathbf{id}_2 \$$ | reduce by $T \rightarrow F$ |
| $\$ T$ | $* \mathbf{id}_2 \$$ | shift |
| $\$ T *$ | $\mathbf{id}_2 \$$ | shift |
| $\$ T * \mathbf{id}_2$ | $\$$ | reduce by $F \rightarrow \mathbf{id}$ |
| $\$ T * F$ | $\$$ | reduce by $T \rightarrow T * F$ |
| $\$ T$ | $\$$ | reduce by $E \rightarrow T$ |




Tree “grows” when reduction happens

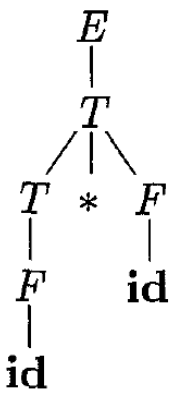
# Shift-Reduce Parsing Example

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow ( E ) \mid \mathbf{id}$$

| STACK | INPUT | ACTION |
|---|---|---|
| $\$$ | $\mathbf{id}_1 * \mathbf{id}_2 \$$ | shift |
| $\$\, \mathbf{id}_1$ | $* \mathbf{id}_2 \$$ | reduce by $F \rightarrow \mathbf{id}$ |
| $\$\, F$ | $* \mathbf{id}_2 \$$ | reduce by $T \rightarrow F$ |
| $\$\, T$ | $* \mathbf{id}_2 \$$ | shift |
| $\$\, T *$ | $\mathbf{id}_2 \$$ | shift |
| $\$\, T * \mathbf{id}_2$ | $\$$ | reduce by $F \rightarrow \mathbf{id}$ |
| $\$\, T * F$ | $\$$ | reduce by $T \rightarrow T * F$ |
| $\$\, T$ | $\$$ | reduce by $E \rightarrow T$ |
| $\$\, E$ | $\$$ | accept |

**Success!!!**




**The final parse tree**

# Shift-Reduce Parsing Example

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

| STACK | INPUT | ACTION |
|---|---|---|
| $\$$ | $\mathbf{id}_1 * \mathbf{id}_2 \$$ | shift |
| $\$\, \mathbf{id}_1$ | $* \mathbf{id}_2 \$$ | reduce by $F \rightarrow \mathbf{id}$ |
| $\$\, F$ | $* \mathbf{id}_2 \$$ | reduce by $T \rightarrow F$ |
| $\$\, T$ | $* \mathbf{id}_2 \$$ | shift |
| $\$\, T *$ | $\mathbf{id}_2 \$$ | shift |
| $\$\, T * \mathbf{id}_2$ | $\$$ | reduce by $F \rightarrow \mathbf{id}$ |
| $\$\, T * F$ | $\$$ | reduce by $T \rightarrow T * F$ |
| $\$\, T$ | $\$$ | reduce by $E \rightarrow T$ |
| $\$\, E$ | $\$$ | accept |

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow ( E ) \mid \mathbf{id}$$

Rightmost derivation:

$E \Rightarrow T$
$\Rightarrow T * F$
$\Rightarrow T * \mathrm{id}$
$\Rightarrow F * \mathrm{id}$
$\Rightarrow \mathrm{id} * \mathrm{id}$

**We can make two observations from the example:**

- Bottom-up parsing is equivalent to finding a rightmost derivation (in reverse).
- At each step, stack + remaining input is a right-sentential form.

# Shift-Reduce Parsing Example

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow ( E ) \mid \mathbf{id}$$

| STACK | INPUT | ACTION |
|---|---|---|
| $\$$ | $\mathbf{id}_1 * \mathbf{id}_2 \$$ | shift |
| $\$ \mathbf{id}_1$ | $* \mathbf{id}_2 \$$ | reduce by $F \rightarrow \mathbf{id}$ |
| $\$ F$ | $* \mathbf{id}_2 \$$ | reduce by $T \rightarrow F$ |
| $\$ T$ | $* \mathbf{id}_2 \$$ | shift |
| $\$ T *$ | $\mathbf{id}_2 \$$ | shift |
| $\$ T * \mathbf{id}_2$ | $\$$ | reduce by $F \rightarrow \mathbf{id}$ |
| $\$ T * F$ | $\$$ | reduce by $T \rightarrow T * F$ |
| $\$ T$ | $\$$ | reduce by $E \rightarrow T$ |
| $\$ E$ | $\$$ | accept |

Why shifting $*$ instead of reducing $T$ ?

Why not reducing $F$ to $T$ ?

**Key decisions:**

1. When to shift? When to reduce?
2. Which production to apply when reducing (there could be multiple possibilities)?

# Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
  - Simple LR (SLR)
  - Canonical LR (CLR)
  - Look-ahead LR (LALR)

# LR Parsing (LR语法分析技术)

- **LR($k$) parsers:** the most prevalent type of bottom-up parsers
  - L: left-to-right scan of the input
  - R: construct a rightmost derivation in reverse
  - $k$: use $k$ input symbols of lookahead in making parsing decisions

- LR(0) and LR(1) parsers are of practical interest
  - When $k \geq 2$, the parser becomes too complex to construct (parsing table will be too huge to manage)

# Advantages of LR Parsers

- Table-driven (like non-recursive LL parsers) and powerful
  - Although it is too much work to construct an LR parser by hand, there are parser generators to construct parsing tables automatically
- LR-parsing is the most general nonbacktracking shift-reduce parsing method known
- LR parsers can be constructed to recognize virtually all programming language constructs for which CFGs can be written
- LR grammars can describe more languages than LL grammars
  - Recall the stringent conditions for a grammar to be LL(1)

# When to Shift/Reduce?

| STACK | INPUT | ACTION |
|---|---|---|
| $\$$ | $\mathbf{id}_1 * \mathbf{id}_2 \$$ | shift |
| $\$\,\mathbf{id}_1$ | $* \mathbf{id}_2 \$$ | reduce by $F \to \mathbf{id}$ |
| $\$\,F$ | $* \mathbf{id}_2 \$$ | reduce by $T \to F$ |
| $\$\,T$ | $* \mathbf{id}_2 \$$ | shift |
| $\$\,T *$ | $\mathbf{id}_2 \$$ | shift |
| $\$\,T * \mathbf{id}_2$ | $\$$ | reduce by $F \to \mathbf{id}$ |
| $\$\,T * F$ | $\$$ | reduce by $T \to T * F$ |
| $\$\,T$ | $\$$ | reduce by $E \to T$ |
| $\$\,E$ | $\$$ | accept |

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid \mathbf{id}$$

Parsing input $\mathbf{id}_1 * \mathbf{id}_2$

How does a shift/reduce parser know that *T* on stack top is a bad choice for reduction (the right action is to shift)?

# LR(0) Items (LR(0) 项)

- An LR parser makes shift-reduce decisions by **maintaining states** to keep track of what have been seen during parsing

- An ***LR(0) item*** (item for short) is a production with a dot at some position of the body, indicating how much we have seen at a given time point in the parsing process
  - $A \rightarrow \cdot XYZ$ $\quad A \rightarrow X \cdot YZ$ $\quad A \rightarrow XY \cdot Z$ $\quad A \rightarrow XYZ \cdot$
  - $A \rightarrow X \cdot YZ$ means: we have just seen on the input a string derivable from $X$ and we hope to see a string derivable from $YZ$ next
  - The production $A \rightarrow \epsilon$ generates only one item $A \rightarrow \cdot$

- **State:** a set of LR(0) items (LR(0) 项集)

# Canonical LR(0) Collection

- One collection of states called the ***canonical* LR(0) collection** (LR(0)项集规范族) provides the basis for constructing a DFA to make parsing decisions

- To construct canonical LR(0) collection for a grammar, we need to define:
  - An augmented grammar (增广文法)
  - Two functions: (1) CLOSURE of item sets (项集闭包) and (2) GOTO

# Augmented Grammar

- Augmenting a grammar $G$ with start symbol $S$
  - Introduce a new start symbol $S'$ to take the role of $S$
  - Add a new production $S' \to S$
- Obviously, $L(G) = L(G')$
- **Benefit:** With the augmentation, acceptance occurs when and only when the parser is about to reduce by $S' \to S$
  - Otherwise, acceptance could occur at many points since there may be multiple $S$-productions in $G$

# Closure of Item Sets

```
SetOfItems CLOSURE(I) {
    J = I;
    repeat
        for ( each item A → α·Bβ in J )
            for ( each production B → γ of G )
                if ( B → ·γ is not in J )
                    add B → ·γ to J;
    until no more items are added to J on one round;
    return J;
}
```

- If $I$ is a set of items for a grammar $G$, then CLOSURE($I$) is the set of items constructed from $I$ by the two steps:
    1. Initially, add every item in $I$ to CLOSURE($I$)
    2. If $A \rightarrow \alpha \cdot B\beta$ is in CLOSURE($I$) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \cdot \gamma$ to CLOSURE($I$), if it is not already there. Apply this rule until no more new items can be added to CLOSURE($I$)

- **Intuition:** $A \rightarrow \alpha \cdot B\beta$ indicates that we hope to see a substring derivable from $B\beta$, which has a prefix derivable from $B$. Therefore, we add items for all $B$-productions.

# Example

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow ( E ) \mid \mathbf{id}$$

- Augmented grammar
  - $E' \rightarrow E$ $\quad E \rightarrow E + T \mid T$ $\quad T \rightarrow T * F \mid F$ $\quad F \rightarrow (E) \mid \mathbf{id}$

- Computing the closure of the item set $\{[E' \rightarrow \cdot\, E]\}$
  - Initially, $[E' \rightarrow \cdot\, E]$ is in the closure
  - Add $[E \rightarrow \cdot\, E + T]$ and $[E \rightarrow \cdot\, T]$ to the closure
  - Add $[T \rightarrow \cdot\, T * F]$ and $[T \rightarrow \cdot\, F]$ to the closure
  - Add $[F \rightarrow \cdot\, (E)]$ and $[F \rightarrow \cdot\, \mathbf{id}]$ and reach fixed point

- $[E' \rightarrow \cdot\, E]$
- $[E \rightarrow \cdot\, E + T]$
- $[E \rightarrow \cdot\, T]$
- $[T \rightarrow \cdot\, T * F]$
- $[T \rightarrow \cdot\, F]$
- $[F \rightarrow \cdot\, (E)]$
- $[F \rightarrow \cdot\, \mathbf{id}]$

# The Function GOTO

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to ( E ) \mid \mathbf{id}$$

- **GOTO($I, X$)**, where $I$ is a set of items and $X$ is a grammar symbol, is defined to be the closure of the set of all items $[A \to \alpha X \cdot \beta]$ where $[A \to \alpha \cdot X\beta]$ is in $I$
  - $CLOSURE(\{[A \to \alpha X \cdot \beta] \mid [A \to \alpha \cdot X\beta] \in I\})$

- **Example:** Computing GOTO$(I, +)$ for $I = \{[E' \to E \cdot], [E \to E \cdot +T]\}$
  - There is only one item $[E \to E \cdot +T]$, in which + follows $\cdot$
  - Then compute the CLOSURE$(\{[E \to E + \cdot T]\})$, which contains:
    - $[E \to E + \cdot T]$
    - $[T \to \cdot T * F]$, $[T \to \cdot F]$
    - $[F \to \cdot (E)]$, $[F \to \cdot \mathbf{id}]$

# Constructing Canonical LR(0) Collection

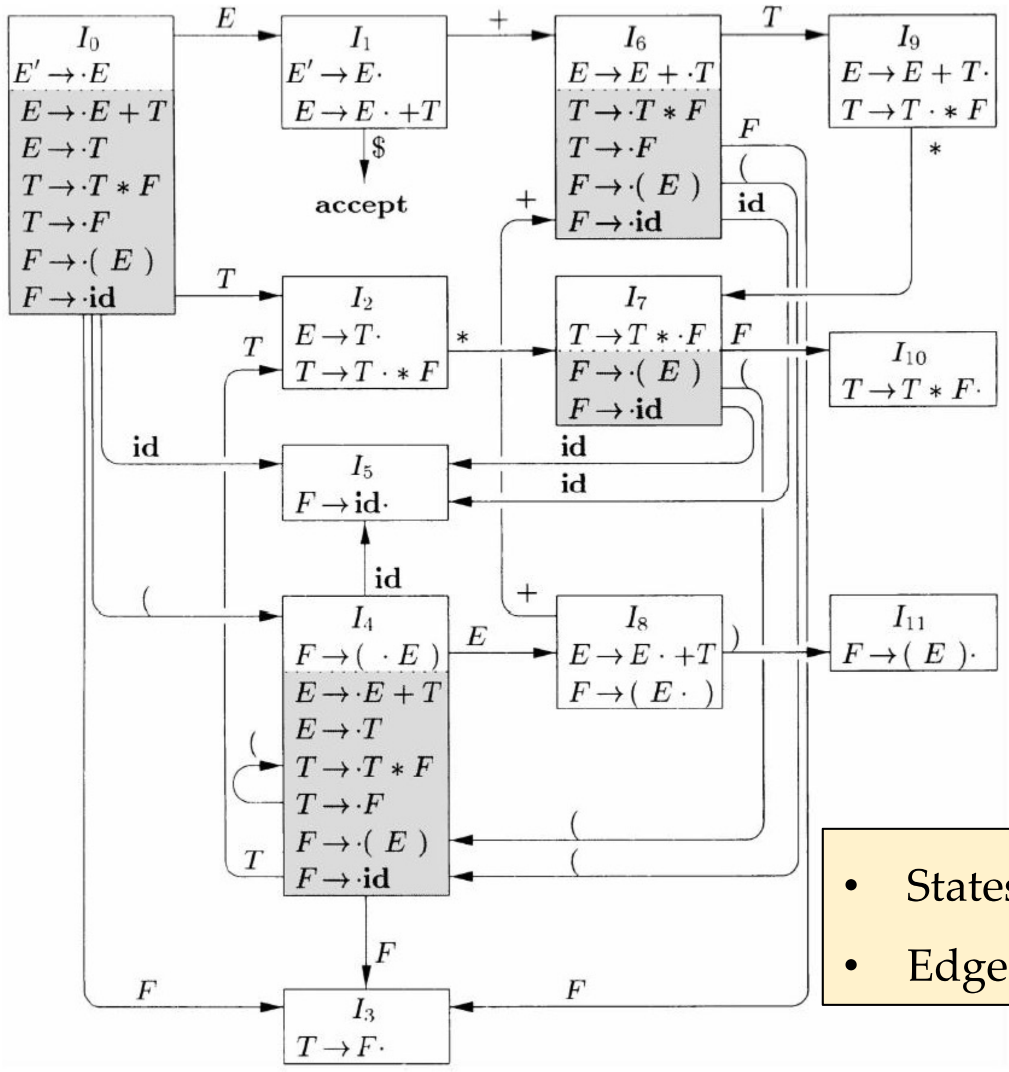
**void** *items*($G'$) {
    $C$ = {CLOSURE({[$S' \to \cdot S$]})};
    **repeat**
        **for** ( each set of items $I$ in $C$ )
            **for** ( each grammar symbol $X$ )
                **if** ( GOTO($I, X$) is not empty and not in $C$ )
                    add GOTO($I, X$) to $C$;
    **until** no new sets of items are added to $C$ on a round;
}

Initially, there is just one item set
(i.e., the initial state)

Iteratively find all possible GOTO targets
(essentially the states in the automaton for parsing)

# Example

The canonical LR(0) collection for the grammar below is $\{I_0, I_1, \ldots, I_{11}\}$




(1) $E \to E + T$

(2) $E \to T$

(3) $T \to T * F$

(4) $T \to F$

(5) $F \to ( E )$

(6) $F \to \mathbf{id}$

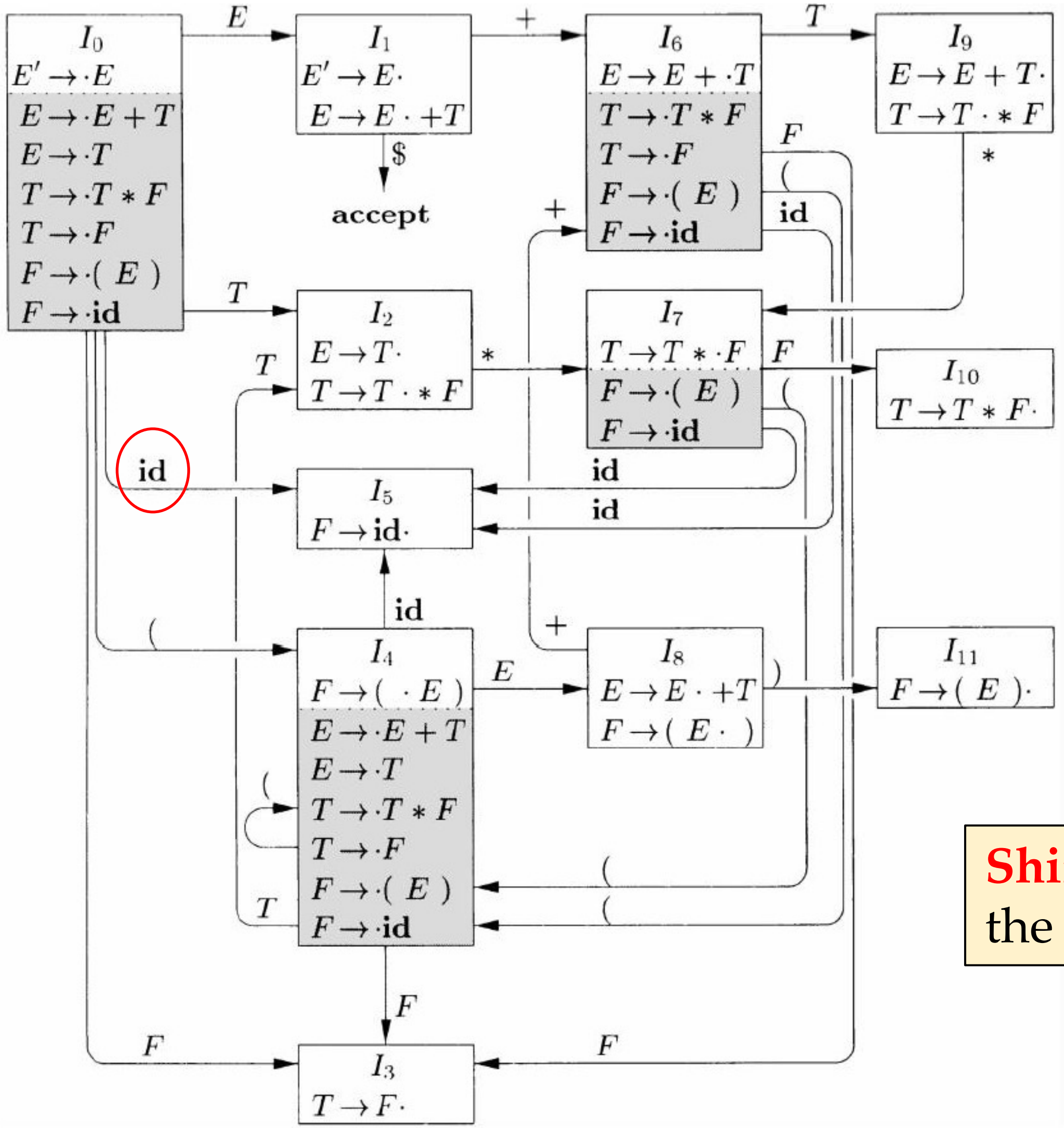
- States are constructed by CLOSURE function
- Edges are constructed by GOTO function

# LR(0) Automaton

- The central idea behind "Simple LR", or SLR, is constructing the LR(0) automaton from the grammar
  - The states are the item sets in the canonical LR(0) collection
  - The transitions are given by the GOTO function
  - The start state is CLOSURE($\{[S' \rightarrow \cdot\, S]\}$)

**LR(0) automaton can effectively help make shift-reduce decisions.**

# Example: Parsing id $*$ id




We only keep states in the stack; grammar symbols can be recovered from the states

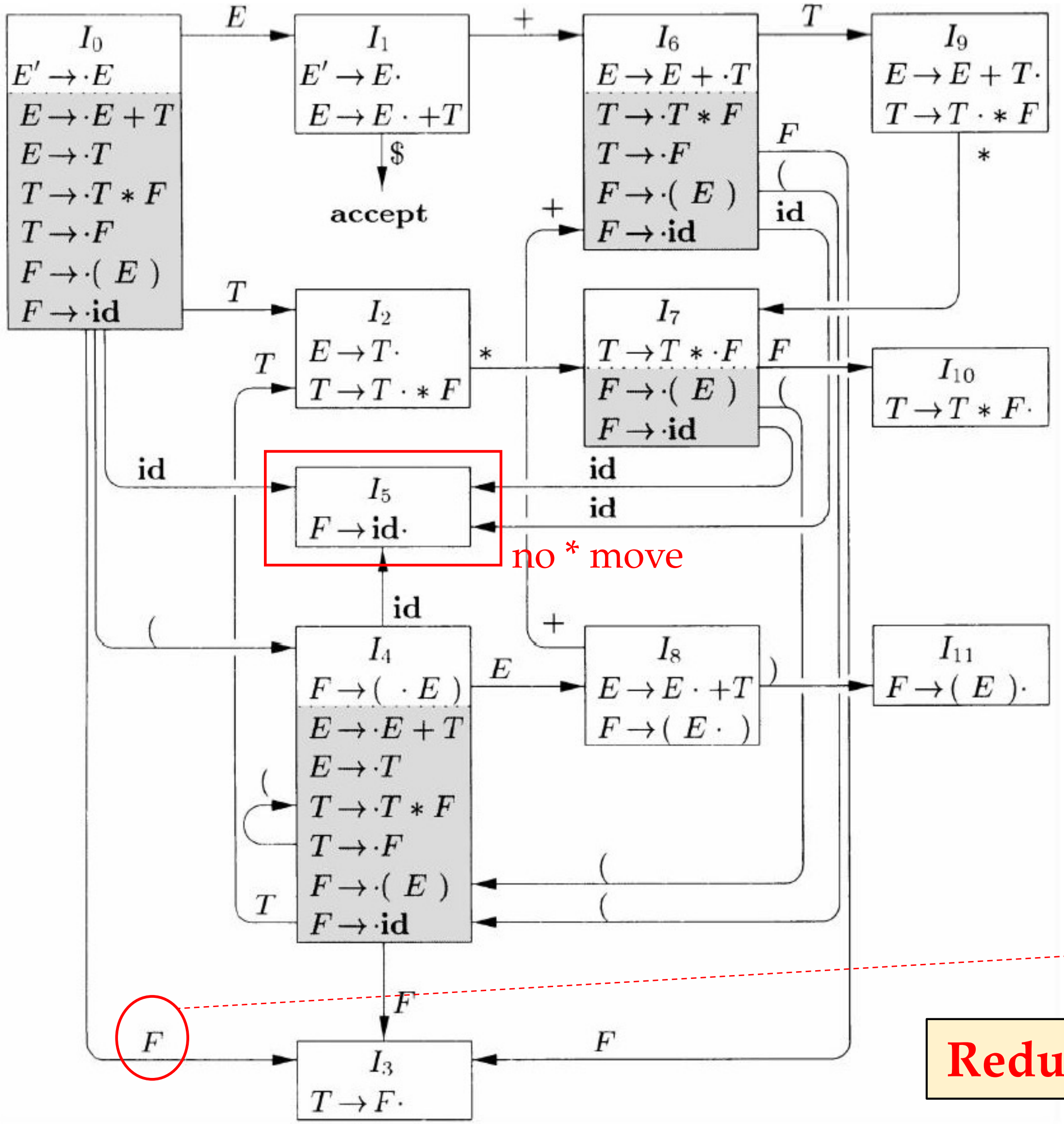
**Stack:** \$ <u>0</u> **Input:** <u>id</u> * id \$

**Grammar Symbols:** \$

**Action:** Shift to 5

**Shift** when the state has a transition on the incoming symbol

# Example: Parsing id $*$ id




We only keep states in the stack; grammar symbols can be recovered from the states

**Stack:** $ 0 5    **Input:** * id $

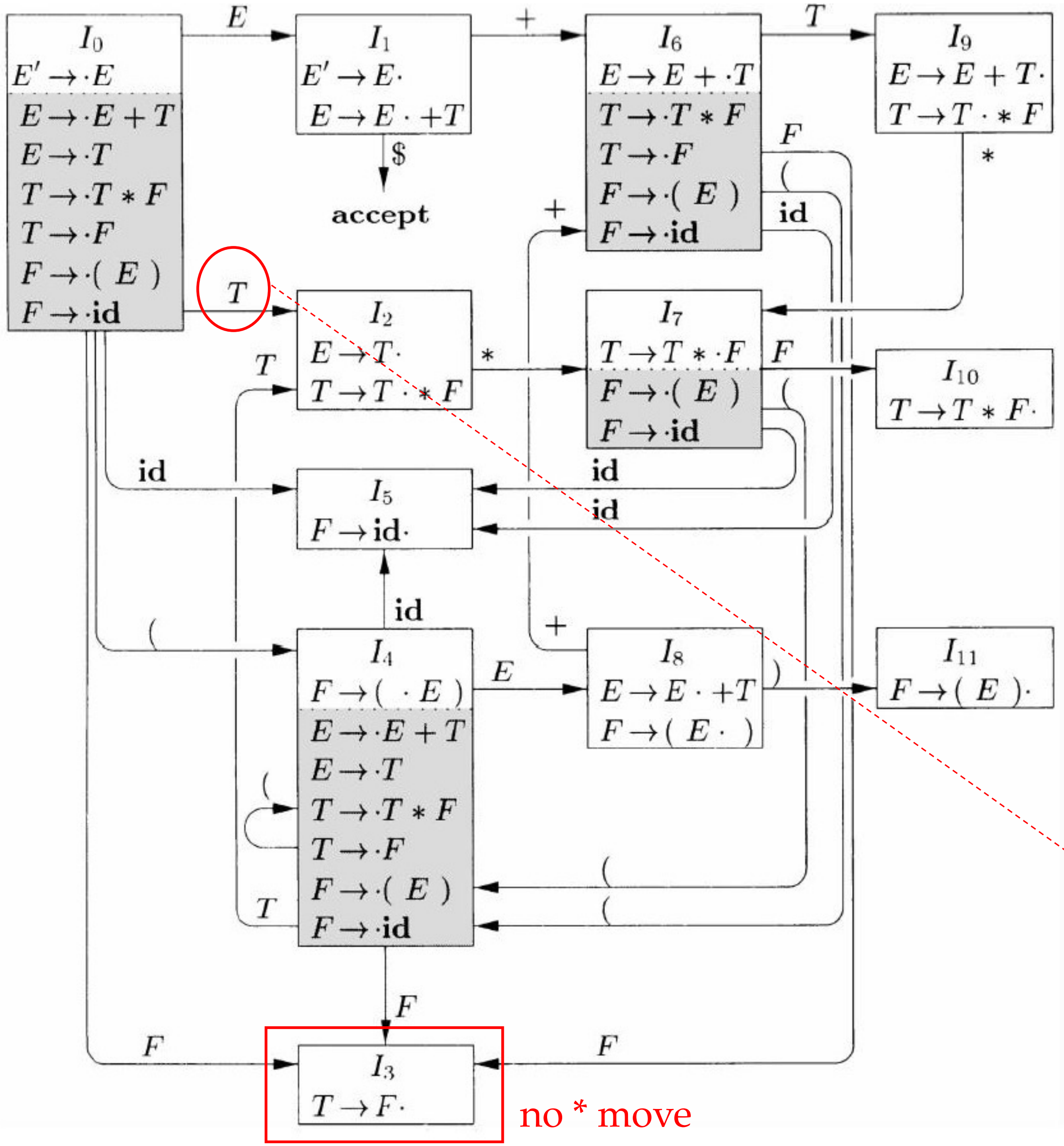
**Grammar Symbols:** $ id

**Action:** Reduce by $F \rightarrow$ **id**

- Pop state 5 (one symbol corresponds to one state)
- Push state 3

**Reduce** when there is no further move

# Example: Parsing id $*$ id




We only keep states in the stack; grammar symbols can be recovered from the states

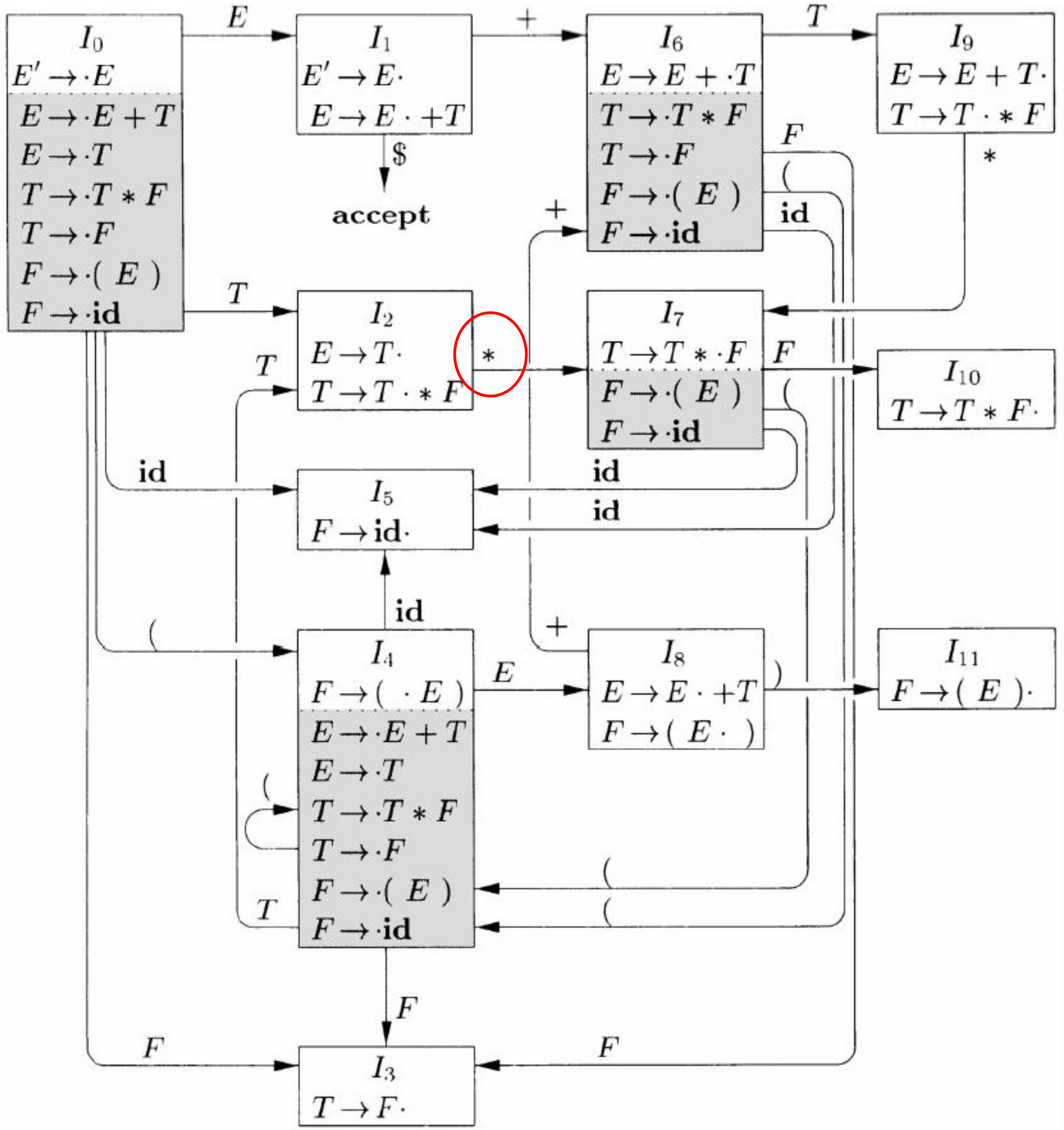
**Stack:** \$ 0 3 **Input:** * id \$

**Grammar Symbols:** \$ $F$

**Action:** Reduce by $T \rightarrow F$

- Pop state 3 (one symbol corresponds to one state)
- Push state 2

# Example: Parsing id $*$ id



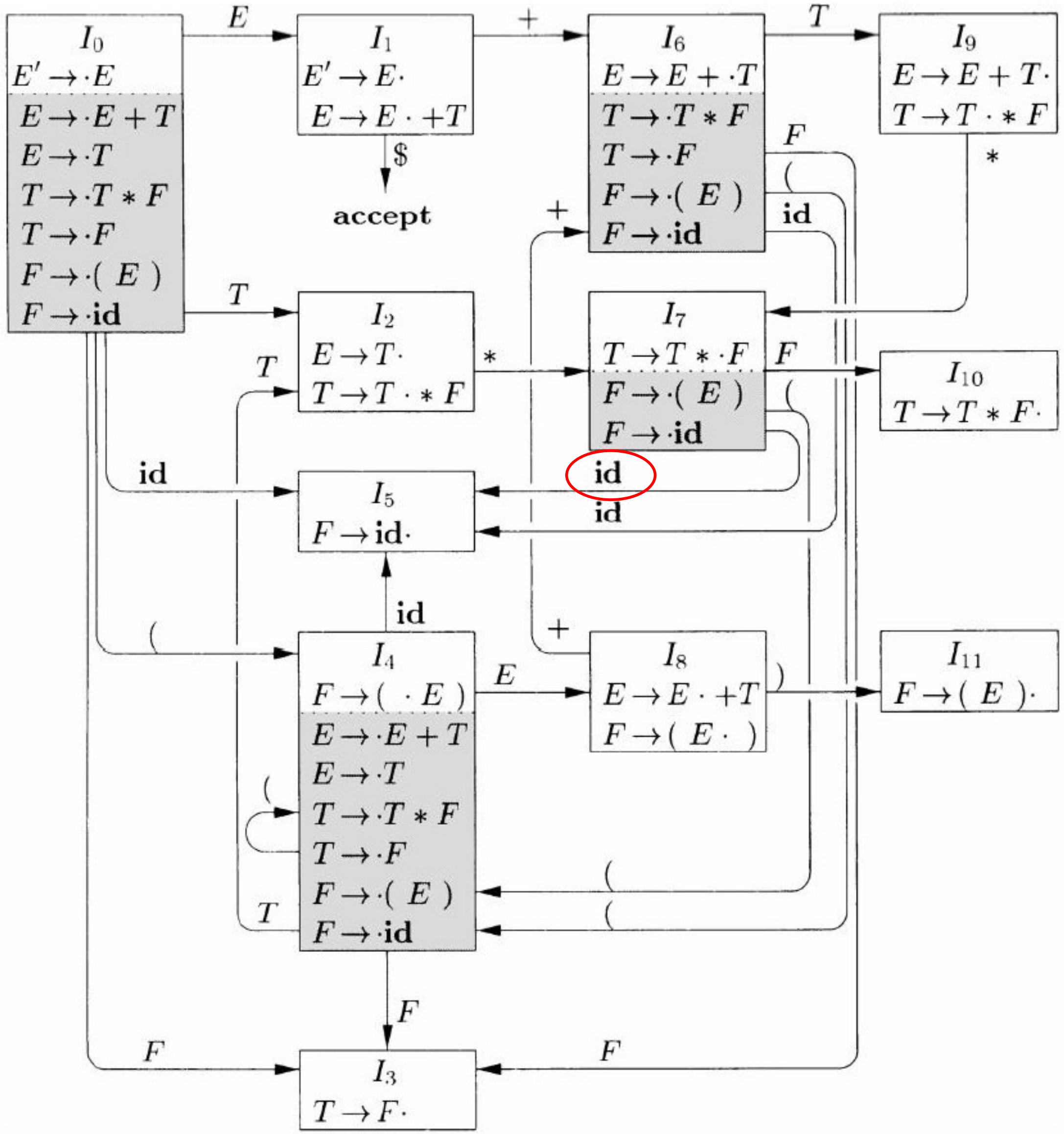

We only keep states in the stack; grammar symbols can be recovered from the states

**Stack:** $ 0 2 **Input:** * id $

**Grammar Symbols:** $ $T$

**Action:** Shift to 7

# Example: Parsing id $*$ id



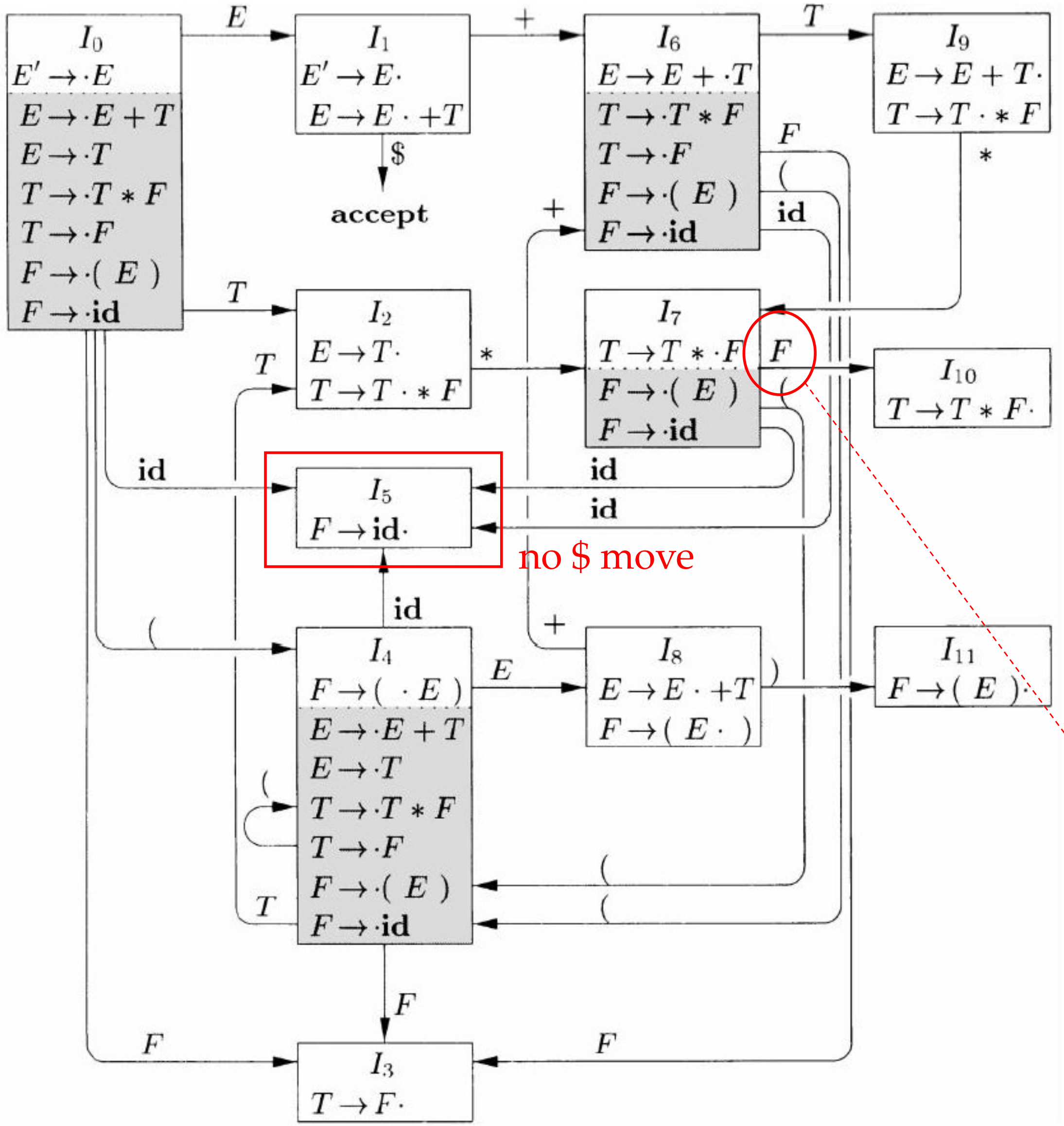

We only keep states in the stack; grammar symbols can be recovered from the states

**Stack:** \$ 0 2 7 **Input:** id \$

**Grammar Symbols:** \$ $T$ *

**Action:** Shift to 5

# Example: Parsing id $*$ id




We only keep states in the stack; grammar symbols can be recovered from the states

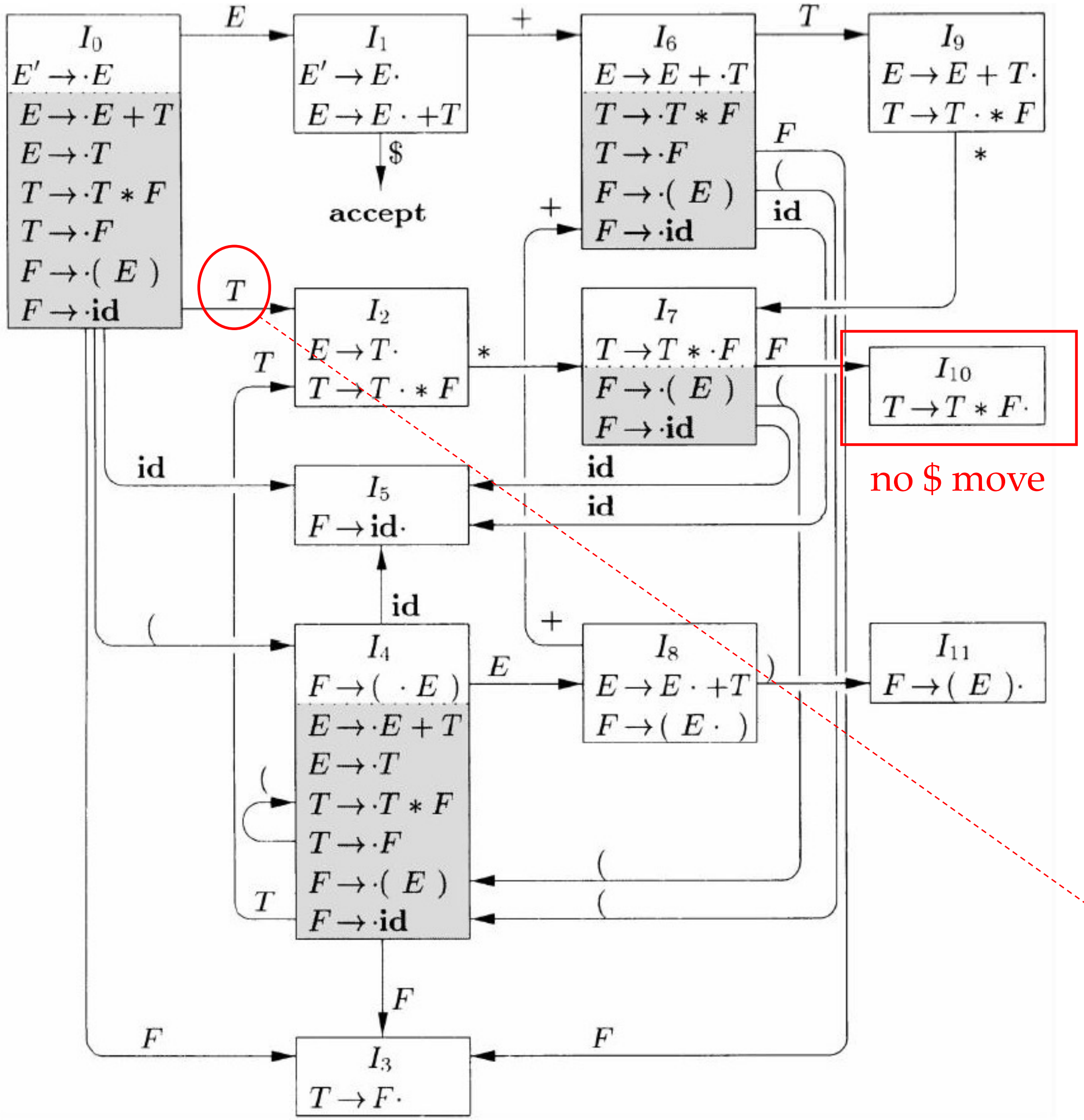
**Stack:** \$ 0 2 7 <u>5</u> **Input:** <u>\$</u>

**Grammar Symbols:** \$ $T$ * id

**Action:** Reduce by $F \to \mathbf{id}$

- Pop state 5 (one symbol corresponds to one state)
- Push state 10

# Example: Parsing id $*$ id




We only keep states in the stack; grammar symbols can be recovered from the states

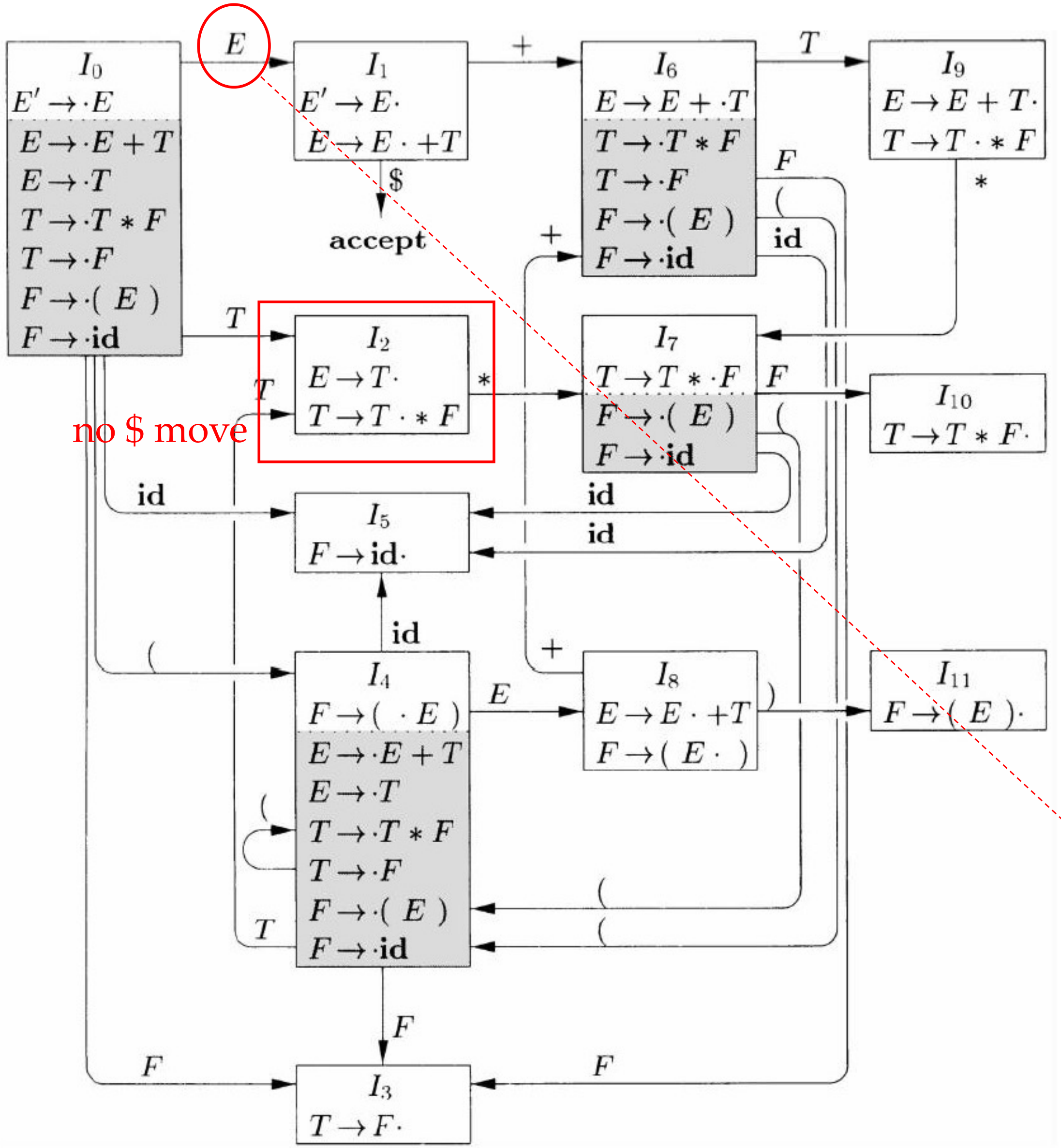
**Stack:** $ 0 2 7 <u>10</u> **Input:** <u>$</u>

**Grammar Symbols:** $ $T$ * $F$

**Action:** Reduce by $T \to T * F$

- Pop states 2, 7, 10 (one symbol corresponds to one state)
- Push state 2

# Example: Parsing **id** $*$ **id**




We only keep states in the stack; grammar symbols can be recovered from the states

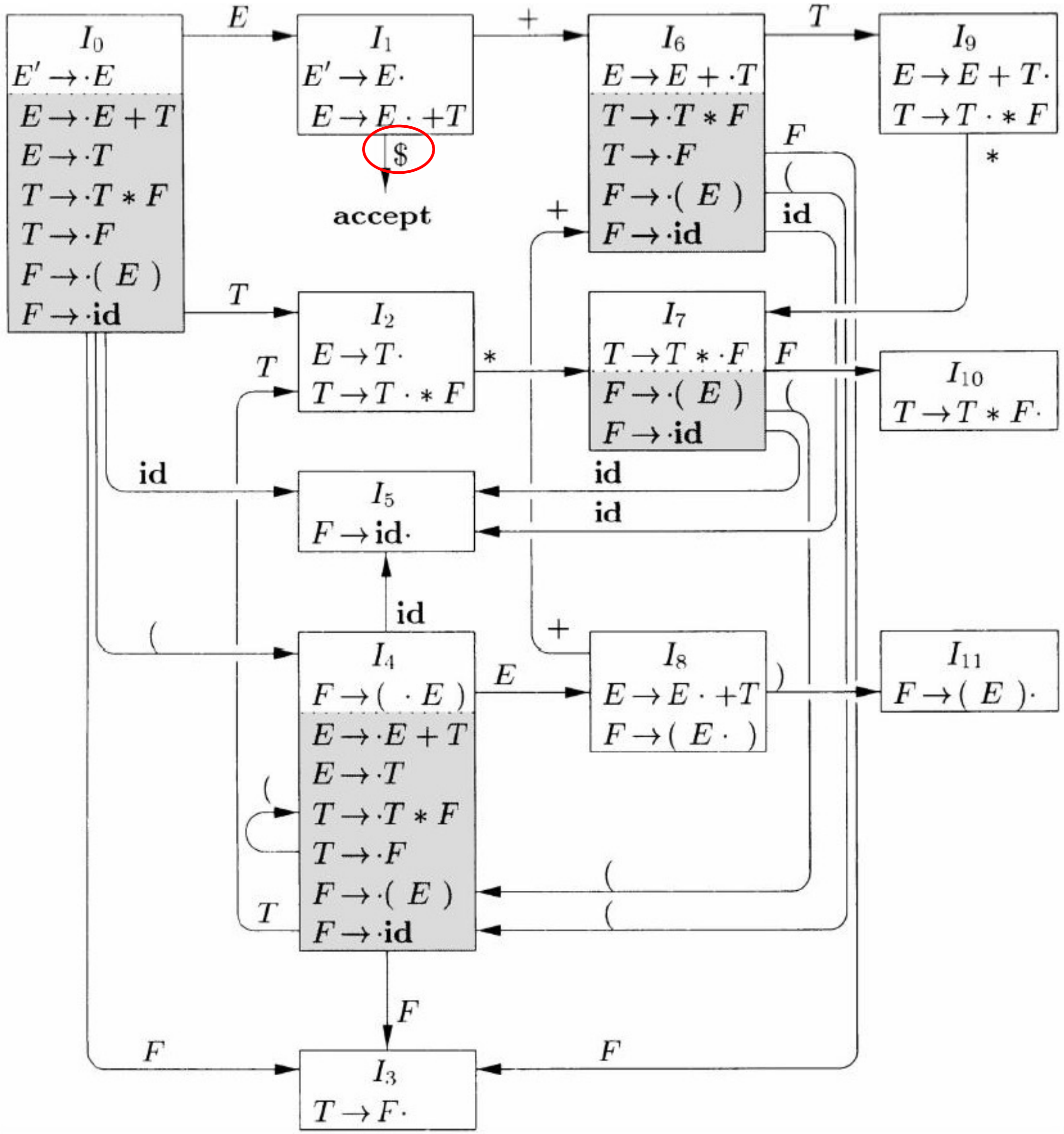
**Stack:** $ 0 <u>2</u>   **Input:** <u>$</u>

**Grammar Symbols:** $ $T$

**Action:** Reduce by $E \to T$

- Pop states 2 (one symbol corresponds to one state)
- Push state 1

# Example: Parsing id $*$ id




We only keep states in the stack; grammar symbols can be recovered from the states

**Stack:** $ 0 <u>1</u> **Input:** <u>$</u>

**Grammar Symbols:** $ $E$

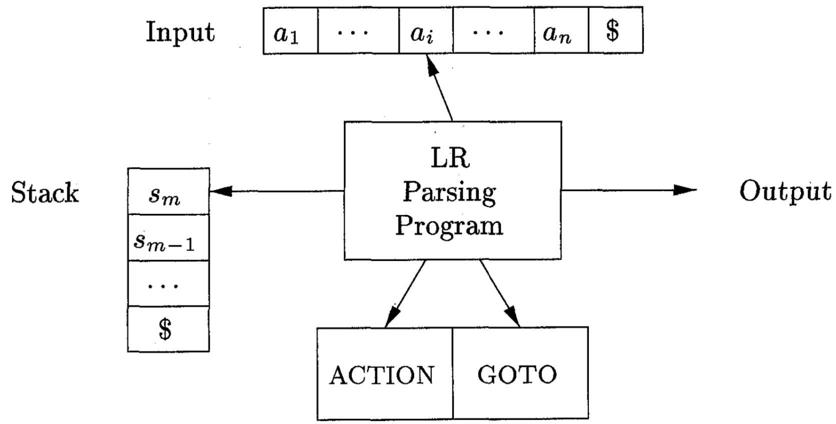
**Action:** Accept

The complete parsing steps:

| LINE | STACK | SYMBOLS | INPUT | ACTION |
|---|---|---|---|---|
| (1) | 0 | $ | **id** $*$ **id** $ | shift to 5 |
| (2) | 0 5 | $ **id** | $*$ **id** $ | reduce by $F \to \mathbf{id}$ |
| (3) | 0 3 | $ $F$ | $*$ **id** $ | reduce by $T \to F$ |
| (4) | 0 2 | $ $T$ | $*$ **id** $ | shift to 7 |
| (5) | 0 2 7 | $ $T *$ | **id** $ | shift to 5 |
| (6) | 0 2 7 5 | $ $T *$ **id** | $ | reduce by $F \to \mathbf{id}$ |
| (7) | 0 2 7 10 | $ $T * F$ | $ | reduce by $T \to T * F$ |
| (8) | 0 2 | $ $T$ | $ | reduce by $E \to T$ |
| (9) | 0 1 | $ $E$ | $ | accept |

# LR Parser Structure

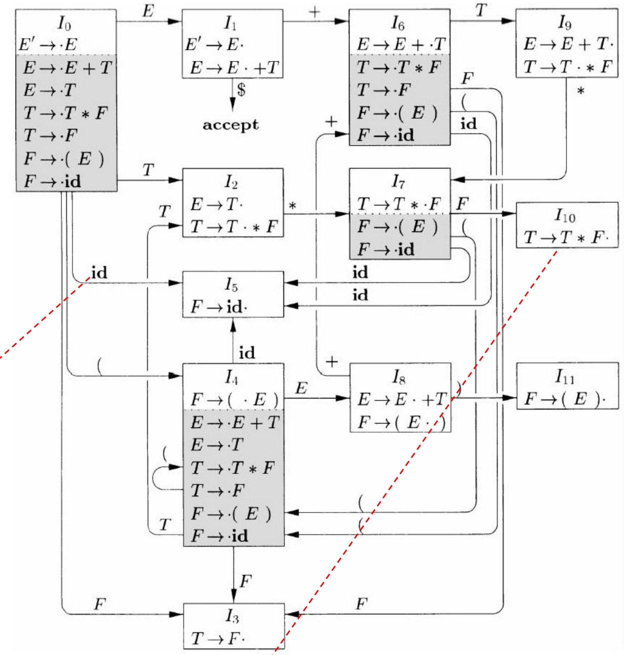
- An LR parser consists of an input, an output, a stack, a driver program, and a parsing table (ACTION + GOTO)
- The driver program is the same for all LR parsers; only the parsing table changes from one parser to another (depending on the parsing algorithm)
- The stack holds a sequence of states
  - In SLR, the stack holds states from the LR(0) automaton
- The parser decides the next action based on (1) the state at the top of the stack and (2) the next terminal read from the input buffer



# Parsing Table: ACTION + GOTO

- The **ACTION** function takes two arguments: (1) a state $i$ and (2) a terminal $a$ (or \$)
- **ACTION[$i$, $a$]** can have one of the four types of values:
  - **Shift $j$:** shift input $a$ to the stack, and uses state $j$ to represent $a$
  - **Reduce $A \rightarrow \beta$:** reduce $\beta$ on the top of the stack to non-terminal $A$
  - **Accept:** The parser accepts the input and finishes parsing
  - **Error:** syntax errors exist
- The **GOTO** function is obtained from the one defined on sets of items: if GOTO($I_i$, $A$) = $I_j$, then GOTO($i$, $A$) = $j$

# Parsing Table Example




| STATE | ACTION | | | | | | GOTO | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | s5 | | | s4 | | | 1 | 2 | 3 |
| 1 | | s6 | | | | acc | | | |
| 2 | | r2 | s7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | s5 | | | s4 | | | 8 | 2 | 3 |
| 5 | | r6 | r6 | | r6 | r6 | | | |
| 6 | s5 | | | s4 | | | | 9 | 3 |
| 7 | s5 | | | s4 | | | | | 10 |
| 8 | | s6 | | | s11 | | | | |
| 9 | | r1 | s7 | | r1 | r1 | | | |
| 10 | | r3 | r3 | | r3 | r3 | | | |
| 11 | | r5 | r5 | | r5 | r5 | | | |

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow ( E )$
(6) $F \rightarrow \mathbf{id}$

- *s*5: shift by pushing state 5 *r*3: reduce using production No. 3
- GOTO entries for terminals are not listed, can be checked in ACTION part

# LR Parser Configurations (态势)

- "**Configuration**" is notation for representing the complete state of the parser (stack status + input status). A *configuration* is a pair:

Stack contents (top on the right) $\underline{(s_0 s_1 \dots s_m,}\ \underline{a_i a_{i+1} \dots a_n \$)}$ Remaining input

- By construction, each state (except $s_0$) in an LR parser corresponds to a set of items and a grammar symbol (the symbol that leads to the state transition, i.e., the symbol on the incoming edge)
  - Suppose $X_i$ is the grammar symbol for state $s_i$
  - Then $X_0 X_1 \dots X_m a_i a_{i+1} \dots a_n$ is a right-sentential form (assume no errors)

# Behavior of the LR Parser

- For the configuration $(s_0 s_1 \dots s_m, a_i a_{i+1} \dots a_n \$)$, the LR parser checks $\mathrm{ACTION}[s_m, a_i]$ in the parsing table to decide the parsing action
  - shift $s$: shift the next state $s$ onto the stack, entering the configuration $(s_0 s_1 \dots s_m s, a_{i+1} \dots a_n \$)$
  - reduce $A \rightarrow \beta$: execute a reduce move, entering the configuration $(s_0 s_1 \dots s_{m-r} s, a_i a_{i+1} \dots a_n \$)$, where $r$ = the length of $\beta$, and $s = \mathrm{GOTO}(s_{m-r}, A)$ ➔ pop $r$ states and push the state $s$ onto stack
  - accept: parsing successful
  - error: the parser has found an error and calls an error recovery routine

# LR-Parsing Algorithm

- **Input:** The parsing table for a grammar $G$ and an input string $\omega$
- **Output:** If $\omega$ is in $L(G)$, the reduction steps of a bottom-up parse for $\omega$; otherwise, an error indication
- **Initial configuration:** $(s_0, \omega\$)$

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

# Constructing SLR-Parsing Tables

- The SLR-parsing table for a grammar $G$ can be constructed based on the LR(0) item sets and LR(0) automaton

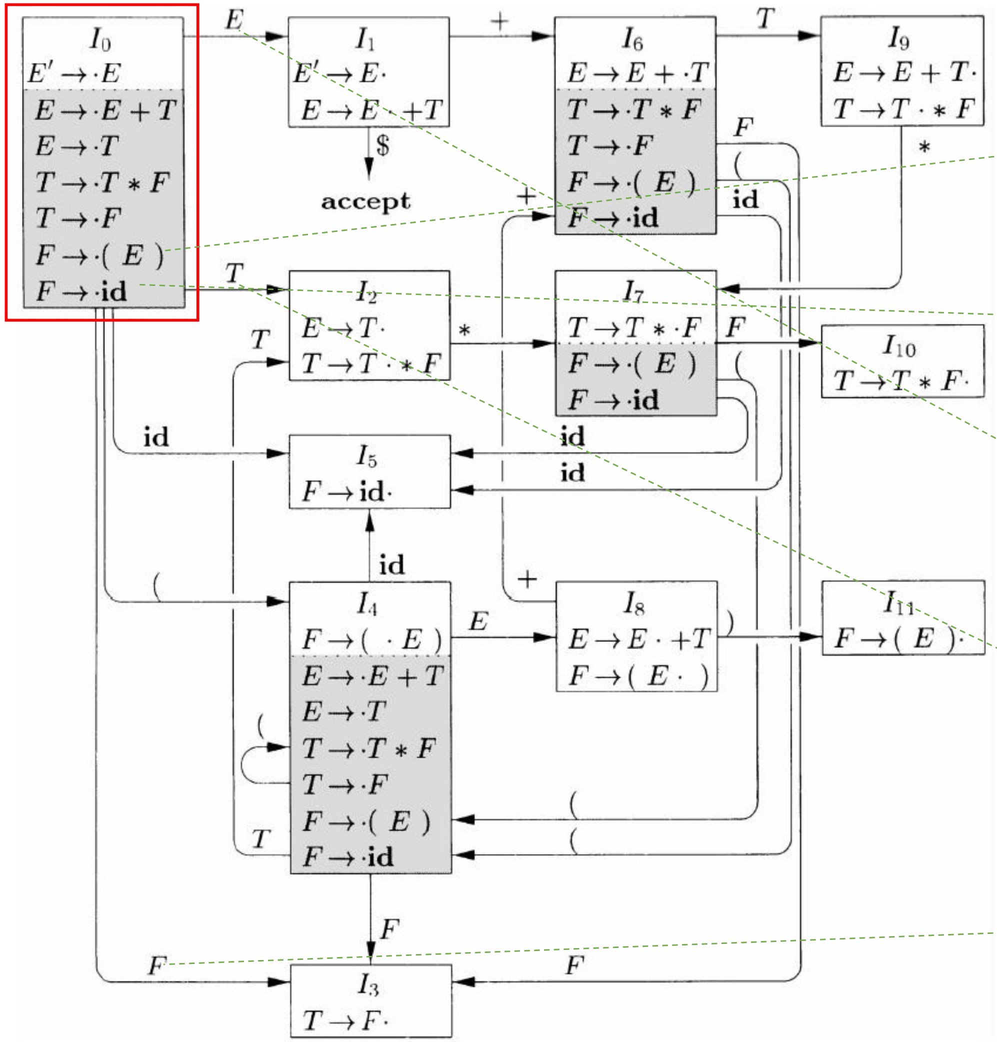
1. Construct the canonical LR(0) collection $\{I_0, I_1, \dots, I_n\}$ for the augmented grammar $G$'
2. State $i$ is constructed from $I_i$. ACTION can be determined as follows:
   - If $[A \to \alpha \cdot a\beta]$ is in $I_i$ and $\mathrm{GOTO}[I_i, a] = I_j$, then set $\mathrm{ACTION}[i, a]$ to "shift $j$" (here $a$ must be a terminal)
   - If $[A \to \alpha \cdot]$ is in $I_i$, then set $\mathrm{ACTION}[i, a]$ to "reduce $A \to \alpha$" for **all $a$ in $\mathrm{FOLLOW}(A)$**; here $A$ may not be $S$'
   - If $[S' \to S \cdot]$ is in $I_i$, then set $\mathrm{ACTION}[i, \$]$ to "accept"
3. The goto transitions for state $i$ are constructed for all nonterminals $A$ using the rule: If $\mathrm{GOTO}(I_i, A) = I_j$, then $\mathrm{GOTO}(i, A) = j$

# Constructing SLR-Parsing Tables

4. All entries not defined in steps 2 and 3 are set to "error"
5. Initial state is the one constructed from the item set containing $[S' \to \cdot S]$

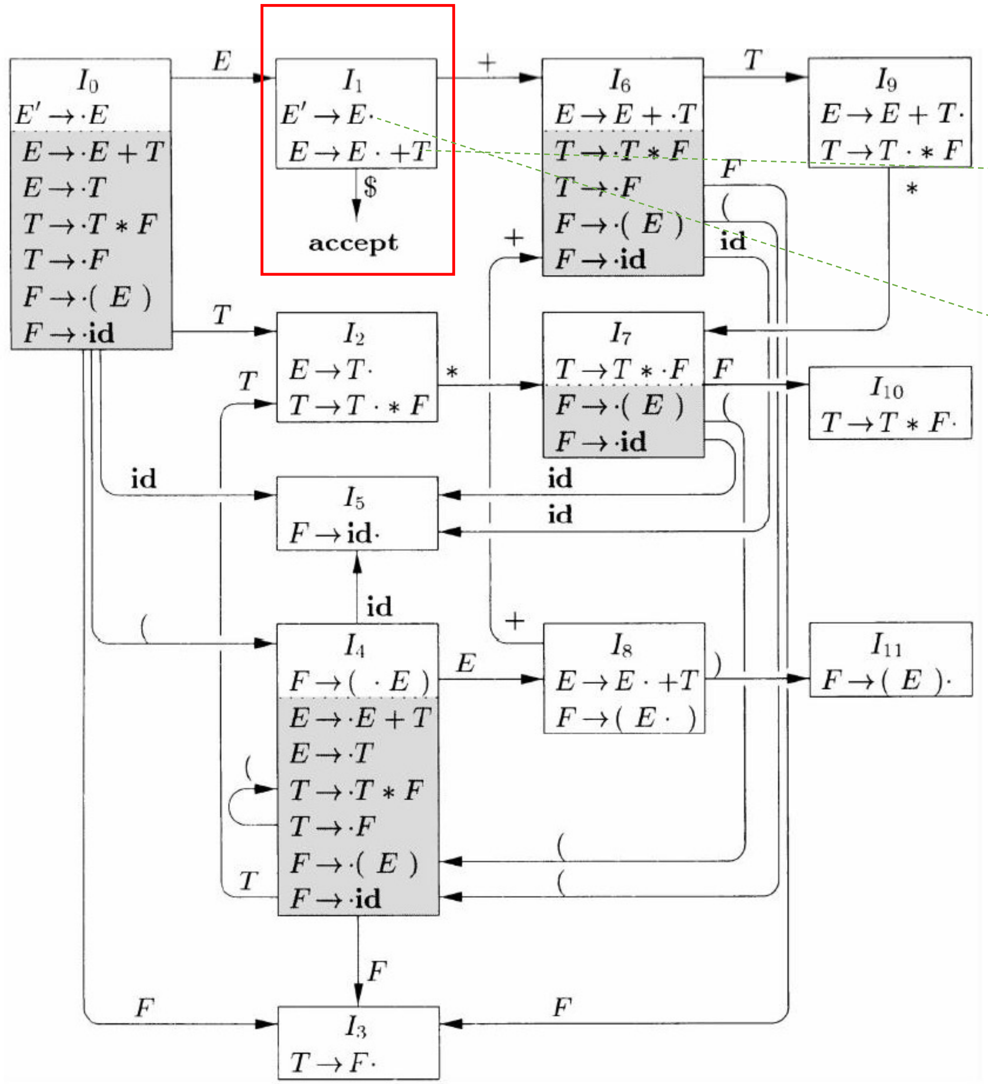
If there is no conflict during the parsing table construction (i.e., multiple actions for a table entry), the grammar is **SLR(1)**

# Example



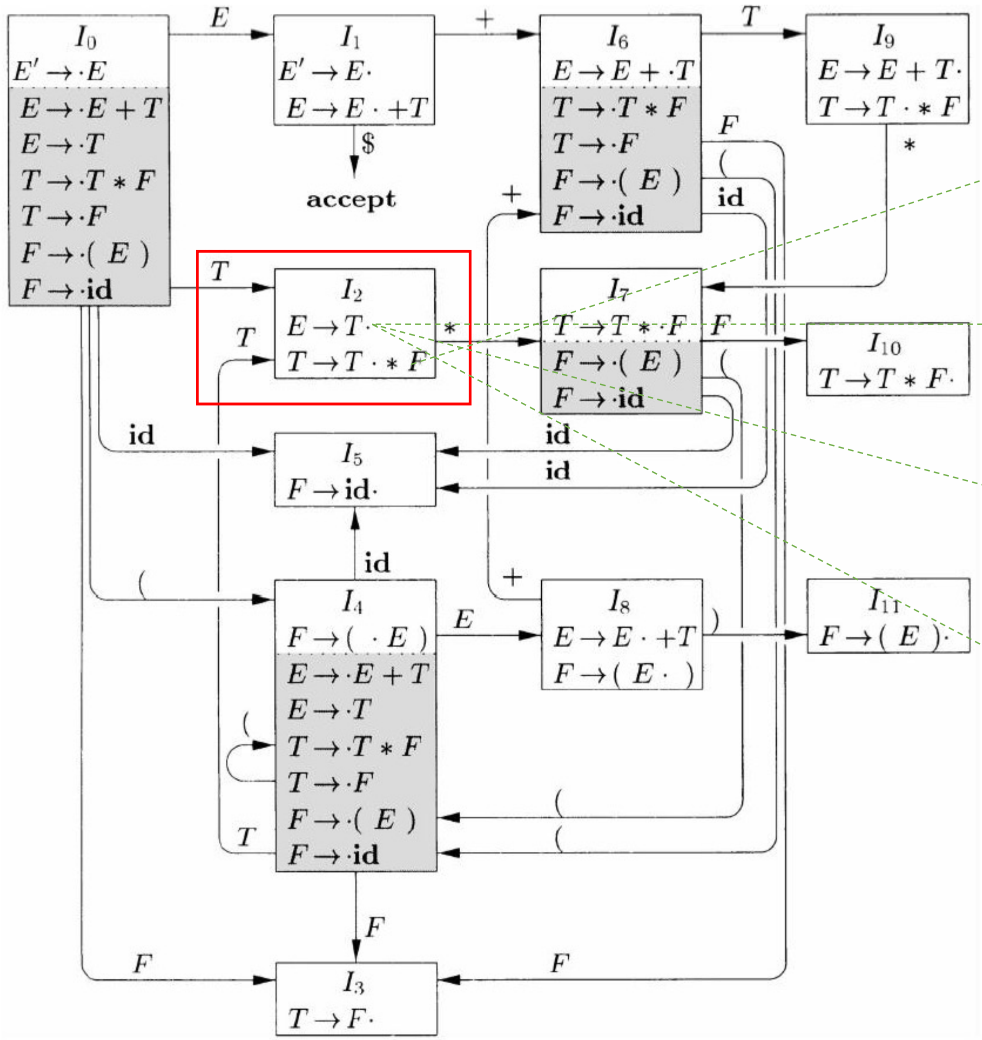

- ACTION(0, () = s4 (shift 4)
- ACTION(0, **id**) = s5
- GOTO$[0, E] = 1$
- GOTO$[0, T] = 2$
- GOTO$[0, F] = 3$

# Example




- $\text{ACTION}(1, +) = \text{s6}$
- $\text{ACTION}(1, \$) = \text{accept}$

# Example




- ACTION(2,∗) = s7
- ACTION(2, $) = reduce $E \to T$
- ACTION(2, +) = reduce $E \to T$
- ACTION(2, )) = reduce $E \to T$

FOLLOW(E) = {$, +, )}

# Non-SLR Grammar

- Grammar
  - $S \rightarrow L = R \mid R$
  - $L \rightarrow * R \mid \mathbf{id}$
  - $R \rightarrow L$
- For item set $I_2$:
  - According to item #1: ACTION$[2, =]$ is "s6"
  - According to item #2: ACTION$[2, =]$ is "reduce $R \rightarrow L$" (FOLLOW$(R)$ contains $=$)

$I_0$: $S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

$I_1$: $S' \rightarrow S \cdot$

$I_2$: $S \rightarrow L \cdot = R$
$R \rightarrow L \cdot$

$I_3$: $S \rightarrow R \cdot$

$I_4$: $L \rightarrow * \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

$I_5$: $L \rightarrow \mathbf{id} \cdot$

$I_6$: $S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

$I_7$: $L \rightarrow * R \cdot$

$I_8$: $R \rightarrow L \cdot$

$I_9$: $S \rightarrow L = R \cdot$

This grammar is not ambiguous

CLR and LALR will succeed on a larger collection of grammars, including the above one. However, there exist unambiguous grammars for which every LR parser construction method will encounter conflicts.

# Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
  - Simple LR (SLR)
  - Canonical LR (CLR)
  - Look-ahead LR (LALR)

# Weakness of the SLR Method

- In SLR, the state $i$ calls for reduction by $A \rightarrow \alpha$ if (1) the item set $I_i$ contains item $[A \rightarrow \alpha \cdot]$ and (2) input symbol $a$ is in $\mathrm{FOLLOW}(A)$

- In some situations, after reduction, the content $\beta\alpha$ on stack top would become $\beta A$ that cannot be followed by $a$ in any right-sentential form* (i.e., only requiring "$a$ is in $\mathrm{FOLLOW}(A)$" is not enough, $\beta A$ cannot be followed by $a$)

* Although SLR algorithm requires $a$ to belong to FOLLOW(A), it is still too casual as the stack content below A is not considered ($\beta$ is not considered).

# Example: Parsing id = id

- $S \to L = R \mid R$
- $L \to * R \mid \mathbf{id}$
- $R \to L$

| | |
|---|---|
| $I_0$: | $S' \to \cdot S$<br>$S \to \cdot L = R$<br>$S \to \cdot R$<br>$L \to \cdot * R$<br>$L \to \cdot\mathbf{id}$<br>$R \to \cdot L$ |
| $I_1$: | $S' \to S\cdot$ |
| $I_2$: | $S \to L\cdot = R$<br>$R \to L\cdot$ |
| $I_3$: | $S \to R\cdot$ |
| $I_4$: | $L \to *\cdot R$<br>$R \to \cdot L$<br>$L \to \cdot * R$<br>$L \to \cdot\mathbf{id}$ |

| | |
|---|---|
| $I_5$: | $L \to \mathbf{id}\cdot$ |
| $I_6$: | $S \to L = \cdot R$<br>$R \to \cdot L$<br>$L \to \cdot * R$<br>$L \to \cdot\mathbf{id}$ |
| $I_7$: | $L \to *R\cdot$ |
| $I_8$: | $R \to L\cdot$ |
| $I_9$: | $S \to L = R\cdot$ |

| Stack | Symbols | Input | Action |
|---|---|---|---|
| $0 | | id = id | Shift 5 |
| $05 | id | = id | Reduce by L→id |
| $02 | L | = id | Suppose reduce by R→L |
| $03 | R | = id | **Error!** |

Cannot shift, cannot reduce since FOLLOW(S) = {$}

**Problem: SLR reduces too casually**

**How to know if a reduction is a good move?**
**Utilize the next input symbol to precisely determine whether to call for a reduction.**

# LR(1) Item

- **Idea:** Carry more information in the state to rule out some invalid reductions (splitting LR(0) states)

- General form of an LR(1) item: $[A \to \alpha \cdot \beta, a]$
  - $A \to \alpha\beta$ is a production and $a$ is a terminal or \$
  - "1" refers to the length of the $2^{\text{nd}}$ component: the *lookahead* (向前看字符)*
  - The lookahead symbol has no effect if $\beta$ is not $\epsilon$ since it only helps determine whether to reduce ($a$ will be inherited during state transitions)
  - An item of the form $[A \to \alpha \cdot, a]$ calls for a reduction by $A \to \alpha$ only if the next input symbol is $a$ (the set of such $a$'s is a **subset** of $\text{FOLLOW}(A)$)

*: LR(0) items do not have lookahead symbols, and hence they are called LR(0)

# Constructing LR(1) Item Sets (1)

- Constructing the collection of LR(1) item sets is essentially the same as constructing the canonical collection of LR(0) item sets. The only differences lie in the CLOSURE and GOTO functions.

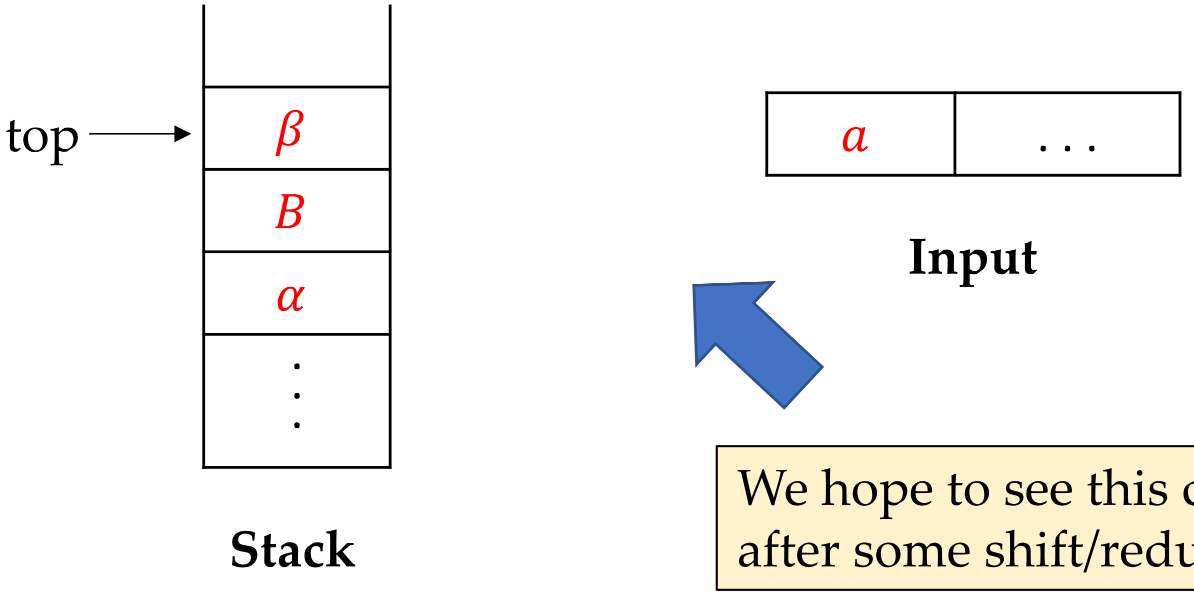
```
SetOfItems CLOSURE(I) {
        J = I;
        repeat
                for ( each item A → α·Bβ in J )
                        for ( each production B → γ of G )
                                if ( B → ·γ is not in J )
                                        add B → ·γ to J;
        until no more items are added to J on one round;
        return J;
}
```

```
SetOfItems CLOSURE(I) {
        repeat
                for ( each item [A → α·Bβ, a] in I )
                        for ( each production B → γ in G' )
                                for ( each terminal b in FIRST(βa) )
                                        add [B → ·γ, b] to set I;
        until no more items are added to I;
        return I;
}
```

It only generates the new item $[B \rightarrow \cdot\, \gamma, b]$ from $[A \rightarrow \alpha \cdot B\beta, a]$ if $b$ is in $\mathrm{FIRST}(\beta a)$

# Why $b$ should be in FIRST($\beta a$)?

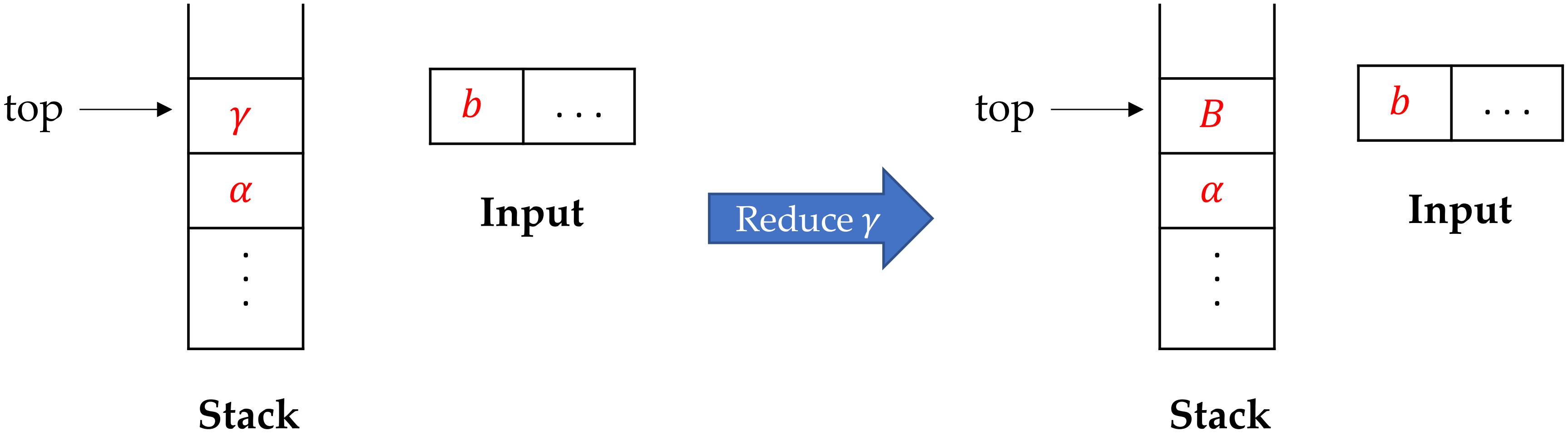
- The item $[A \to \alpha \cdot B\beta, a]$ will derive $[A \to \alpha B\beta \cdot, a]$, which calls for reduction when the stack top contains $\alpha B\beta$ and the next input symbol is $a$




We hope to see this configuration after some shift/reduce steps.

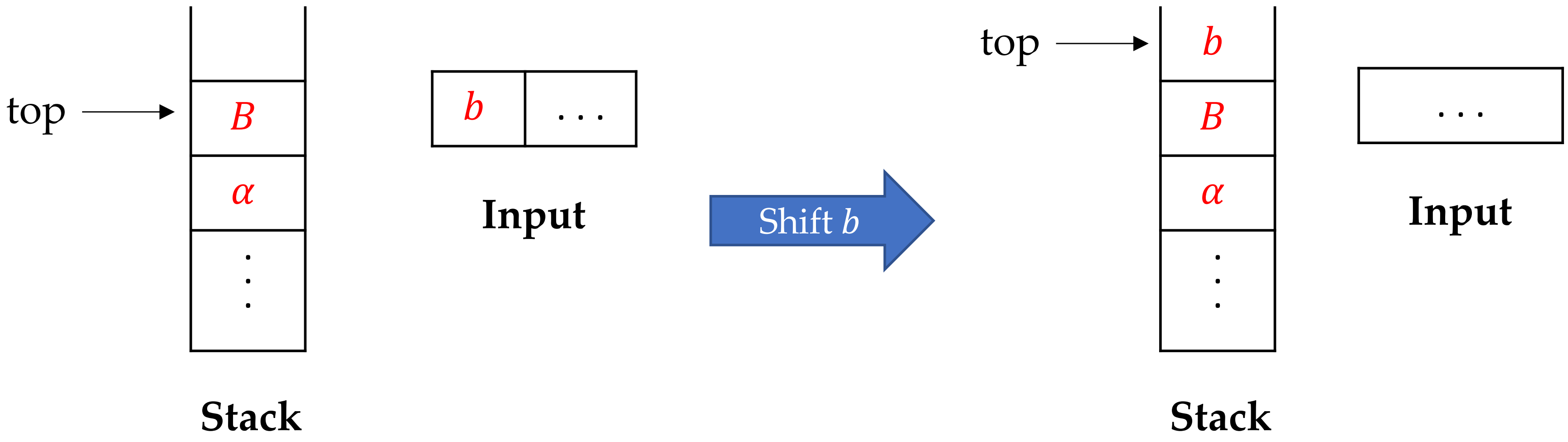
# Why $b$ should be in FIRST($\beta a$)?

- When generating the item $[B \rightarrow \cdot \gamma, b]$ from $[A \rightarrow \alpha \cdot B\beta, a]$, suppose we allow that $b$ is not in $FIRST(\beta a)$
- We add the item $[B \rightarrow \cdot \gamma, b]$ because we hope that at certain time point during parsing, when we see $\gamma$ on stack top and $b$ as the next input symbol, we can first reduce $\gamma$ to $B$ so that in some later step the stack top would contain $\alpha B\beta$ (then we can further reduce it to $A$)



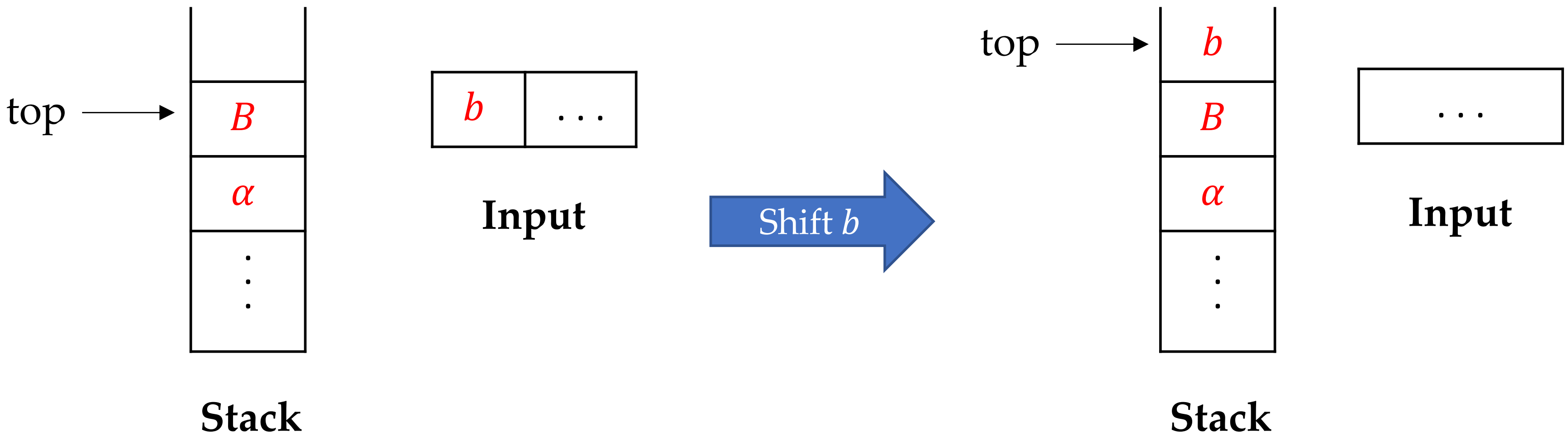
# Why $b$ should be in FIRST($\beta a$)?

- If we reduce $\gamma$ to B, the next action would be "shift $b$ to the stack"
  - Because the production $A \rightarrow \alpha B \beta$ tells us that we are ready for reduction only when we see $\alpha B \beta$ on stack top (i.e., "the next action is shift" is guaranteed by design, as we want to eventually see $\alpha B \beta$ on stack top)



# Why $\boldsymbol{b}$ should be in FIRST$(\boldsymbol{\beta a})$?

- Since $b$ is not in $FIRST(\beta a)$, the stack top will never become the form $\alpha B\beta$, which means we will never be able to reduce $\alpha B\beta$ to $A$
- Then why should we generate $[B \rightarrow\cdot \gamma, b]$ from $[A \rightarrow \alpha \cdot B\beta, a]$ in the first palce???



# Constructing LR(1) Item Sets (2)

- Constructing the collection of LR(1) item sets is essentially the same as constructing the canonical collection of LR(0) item sets. The only differences lie in the CLOSURE and GOTO functions.

```
SetOfItems GOTO(I, X) {
    initialize J to be the empty set;
    for ( each item [A → α·Xβ, a] in I )
        add item [A → αX·β, a] to set J;
    return CLOSURE(J);
}
```

**GOTO$(I, X)$ in LR(0) item sets:**

The closure of the set of all items $[A \to \alpha X \cdot \beta]$ where $[A \to \alpha \cdot X\beta]$ is in $I$.

The lookahead symbols are passed to new items from existing items

# Constructing LR(1) Item Sets (3)

```
void items(G') {
    C = {CLOSURE({[S' → ·S]})};
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C on a round;
}
```

Constructing the collection of LR(0) item sets

```
void items(G') {
    initialize C to {CLOSURE({[S' → ·S, $]})};
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C;
}
```

Constructing the collection of LR(1) item sets

# LR(1) Item Sets Example

It only generates the new item $[B \rightarrow \cdot\, \gamma, b]$ from $[A \rightarrow \alpha \cdot B\beta, a]$ if $b$ is in FIRST($\beta a$)

- Augmented grammar:
  - $S' \rightarrow S \qquad S \rightarrow CC \qquad C \rightarrow cC \mid d$

- Constructing $I_0$ item set and GOTO function:
  - $I_0 = \text{CLOSURE}([S' \rightarrow \cdot\, S, \$]) =$
    - $\{[S' \rightarrow \cdot\, S, \$], [S \rightarrow \cdot\, CC, \$], [C \rightarrow \cdot\, cC, c/d], [C \rightarrow \cdot\, d, c/d]\}$
  - $\text{GOTO}(I_0, S) = \text{CLOSURE}(\{[S' \rightarrow S \cdot, \$]\}) = \{[S' \rightarrow S \cdot, \$]\}$
  - $\text{GOTO}(I_0, C) = \text{CLOSURE}(\{[S \rightarrow C \cdot C, \$]\}) =$
    - $\{[S \rightarrow C \cdot C, \$], [C \rightarrow \cdot\, cC, \$], [C \rightarrow \cdot\, d, \$]\}$
  - $\text{GOTO}(I_0, c) = \text{CLOSURE}(\{[C \rightarrow c \cdot C, c/d]\}) =$
    - $\{[C \rightarrow c \cdot C, c/d], [C \rightarrow \cdot\, cC, c/d], [C \rightarrow \cdot\, d, c/d]\}$
  - $\text{GOTO}(I_0, d) = \text{CLOSURE}(\{[C \rightarrow d \cdot, c/d]\}) = \{[C \rightarrow d \cdot, c/d]\}$

FIRST($) = {$}

FIRST(C$) = {c, d}

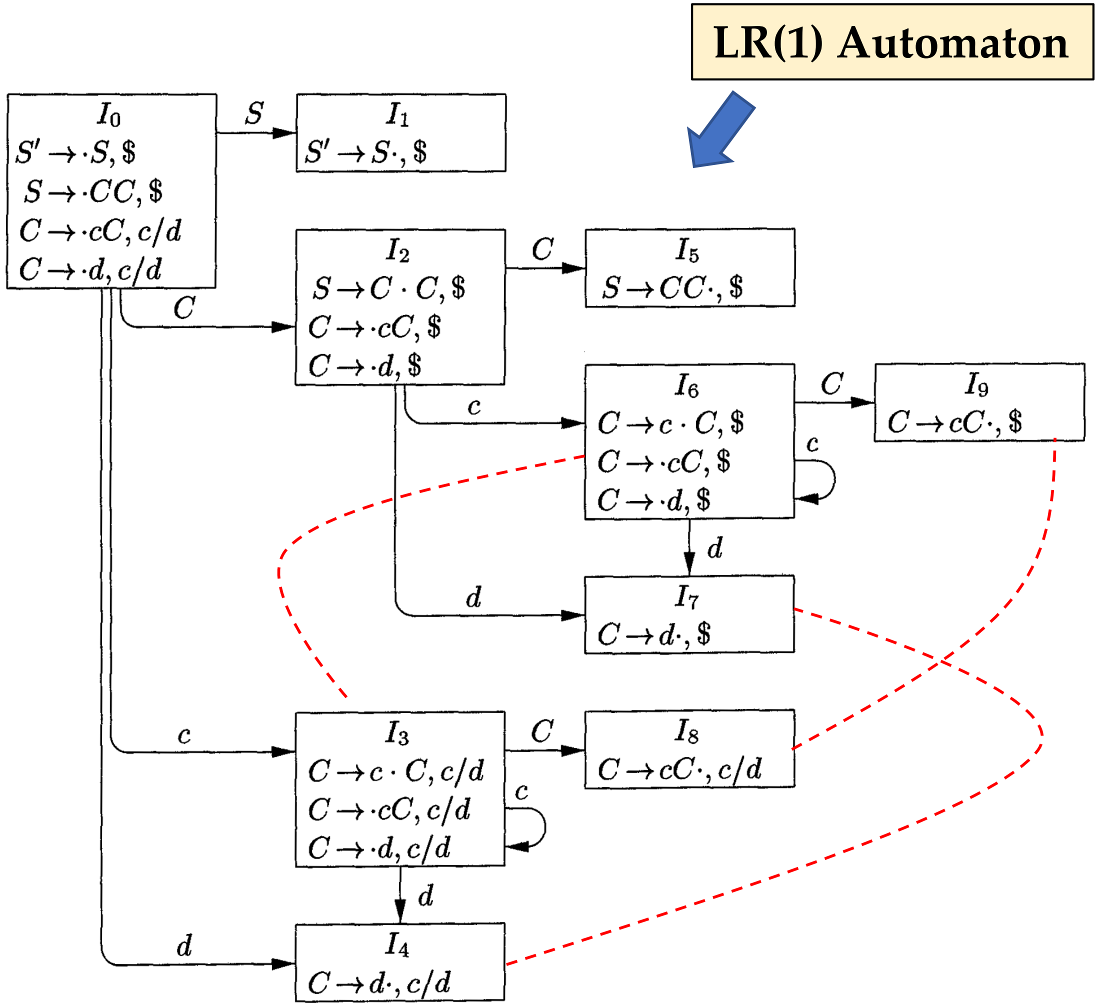
FIRST($) = {$}

# The GOTO Graph Example

**10 states in total**

These states are equivalent if we ignore the lookahead symbols (SLR makes no such distinctions of states):

- $I_3$ and $I_6$
- $I_4$ and $I_7$
- $I_8$ and $I_9$



# Constructing Canonical LR(1) Parsing Tables

1. Construct $C' = \{I_0, I_1, \ldots, I_n\}$, the collection of LR(1) item sets for the augmented grammar $G'$

2. State $i$ of the parser is constructed from $I_i$. Its parsing action is determined as follows:
   - If $[A \to \alpha \cdot a\beta, b]$ is in $I_i$ and $\mathrm{GOTO}(I_i, a) = I_j$, then set $\mathrm{ACTION}[i, a]$ to "*shift j*." Here, $a$ must be a terminal.
   - If $[A \to \alpha \cdot, a]$ is in $I_i$, $A \neq S'$, then set $\mathrm{ACTION}[i, a]$ to "*reduce* $A \to \alpha$"
   - If $[S' \to S \cdot, \$]$ is in $I_i$, then set $\mathrm{ACTION}[i, \$]$ to "*accept*"

More restrictive than SLR

If any conflicting actions result from the above rules, we say the grammar is not LR(1)

# Constructing Canonical LR(1) Parsing Tables

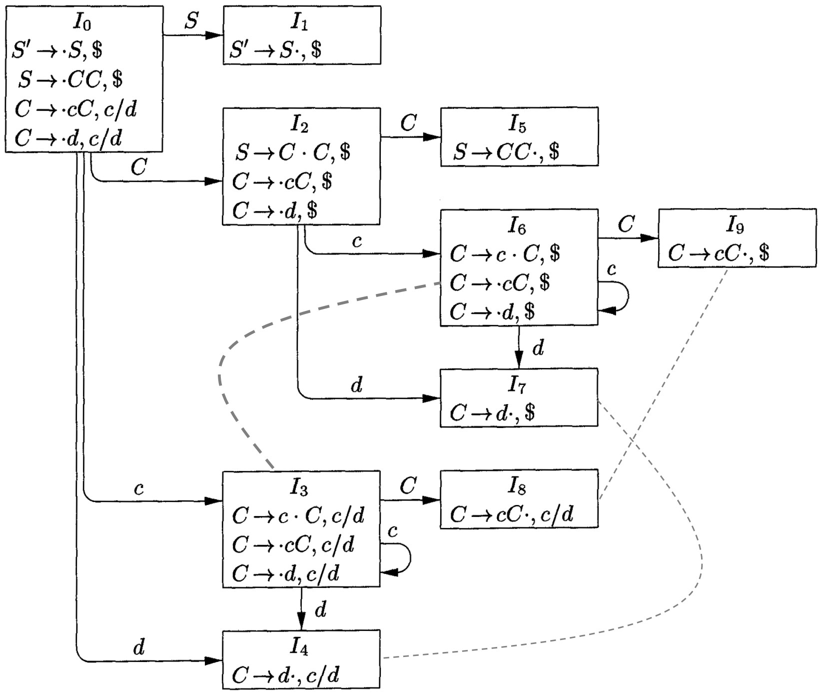
3. The goto transitions for state $i$ are constructed from all nonterminals $A$ using the rule: If $\mathtt{GOTO}(I_i, A) = I_j$, then $\mathtt{GOTO}(i, A) = j$

4. All entries not defined in steps (2) and (3) are made "error"

5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot\, S, \$]$

# LR(1) Parsing Table Example

**Grammar:** $S' \rightarrow S$ $\quad S \rightarrow CC$ $\quad C \rightarrow cC \mid d$




| STATE | ACTION | | | GOTO | |
|---|---|---|---|---|---|
| | $c$ | $d$ | $\$$ | $S$ | $C$ |
| 0 | s3 | s4 | | 1 | 2 |
| 1 | | | acc | | |
| 2 | s6 | s7 | | | 5 |
| 3 | s3 | s4 | | | 8 |
| 4 | r3 | r3 | | | |
| 5 | | | r1 | | |
| 6 | s6 | s7 | | | 9 |
| 7 | | | r3 | | |
| 8 | r2 | r2 | | | |
| 9 | | | r2 | | |

Three pairs of states can be seen as being split from the corresponding LR(0) states:

(3, 6) (4, 7) (8, 9)

# Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
  - Simple LR (SLR)
  - Canonical LR (CLR)
  - Look-ahead LR (LALR)

# Lookahead LR (LALR) Method

- SLR(1) is not powerful enough to handle a large collection of grammars (recall the previous unambiguous grammar)

- LR(1) has a huge set of states in the parsing table (states are too fine-grained)

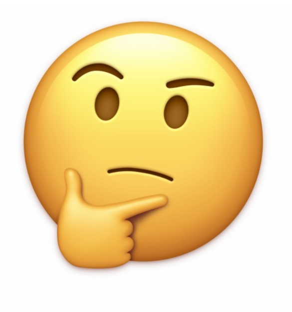
- LALR(1) is often used in practice
  - Keeps the lookahead symbols in the items
  - Its number of states is the same as that of SLR(1)
  - Can deal with most common syntactic constructs of modern programming languages

# Merging States in LR(1) Parsing Tables

**Grammar:**
$S' \rightarrow S$ $\quad S \rightarrow CC$ $\quad C \rightarrow cC \mid d$

- **State 4:**
  - Reduce by $C \rightarrow d$ if the next input symbol is $c$ or $d$
  - Error if \$
- **State 7:**
  - Reduce by $C \rightarrow d$ if the next input symbol is \$
  - Error if $c$ or $d$

| STATE | ACTION | | | GOTO | |
|---|---|---|---|---|---|
| | $c$ | $d$ | \$ | $S$ | $C$ |
| 0 | s3 | s4 | | 1 | 2 |
| 1 | | | acc | | |
| 2 | s6 | s7 | | | 5 |
| 3 | s3 | s4 | | | 8 |
| 4 | r3 | r3 | | | |
| 5 | | | r1 | | |
| 6 | s6 | s7 | | | 9 |
| 7 | | | r3 | | |
| 8 | r2 | r2 | | | |
| 9 | | | r2 | | |



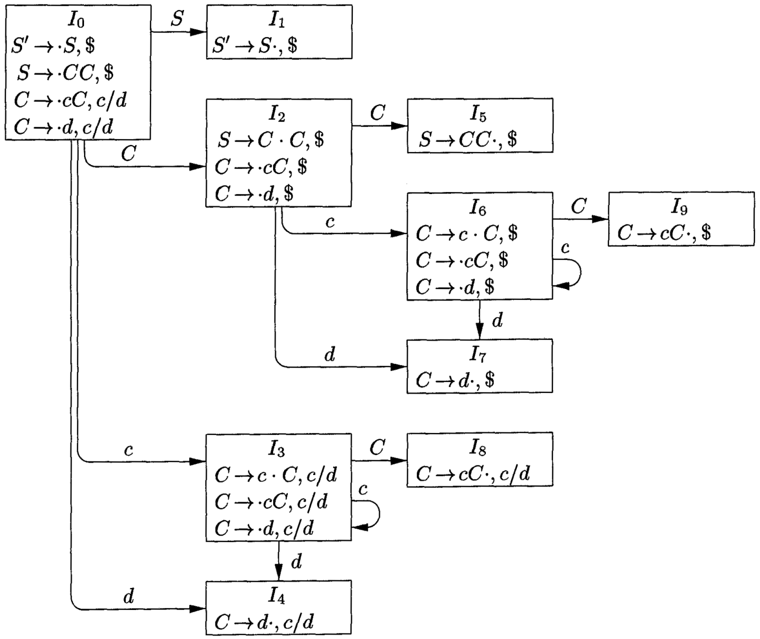
Can we merge states 4 and 7 so that the parser can reduce for all input symbols?

$I_{47}$: $C \rightarrow d \cdot, c/d/\$$

- $I_4$:$[C \rightarrow d \cdot, c/d]$
- $I_7$:$[C \rightarrow d \cdot, \$]$

# The Basic Idea of LALR

- Look for sets of LR(1) items with the same ***core***
  - The <u>core</u> of an LR(1) item set is the set of <u>the first components</u>
    - The core of $I_4$ and $I_7$ is $\{[C \to d\ \cdot]\}$
    - The core of $I_3$ and $I_6$ is $\{[C \to c \cdot C], [C \to \cdot\ cC], [C \to \cdot\ d]\}$

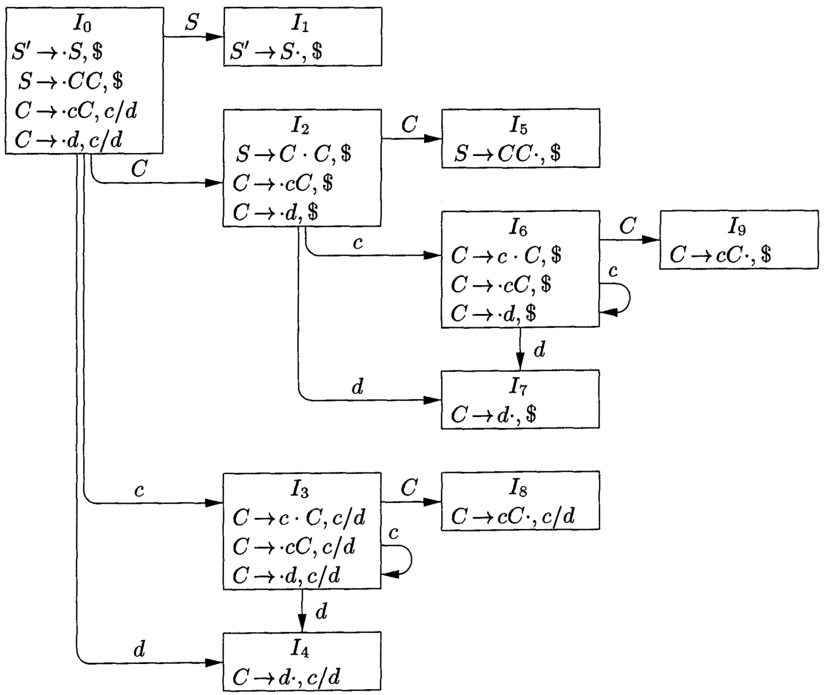


# The Basic Idea of LALR Cont.

- Look for sets of LR(1) items with the same ***core***
  - The core of an LR(1) item set is the set of the first components
    - The core of $I_4$ and $I_7$ is $\{[C \to d \cdot]\}$
    - The core of $I_3$ and $I_6$ is $\{[C \to c \cdot C], [C \to \cdot cC], [C \to \cdot d]\}$
  - In general, a core is a set of LR(0) items
- We may merge the LR(1) item sets with common cores into one set of items

# The Basic Idea of LALR Cont.

- Since the core of $\texttt{GOTO}(I, X)$ depends only on the core of $I$, the goto targets of merged sets also have the same core and hence can be merged




**Consider $I_3$ and $I_6$:**

- The core $\{[C \to c \cdot C], [C \to \cdot\ cC], [C \to \cdot\ d]\}$ determines state transition targets
- Before merging, GOTO($I_3$, $C$) = $I_9$, GOTO($I_6$, $C$) = $I_8$
- After merging, $I_3$ and $I_6$ become $I_{36}$, $I_8$ and $I_9$ become $I_{89}$, and GOTO($I_{36}$, $C$) = $I_{89}$

# Conflicts Caused by State Merging

- Merging states in an LR(1) parsing table may cause conflicts
- Merging does not cause shift/reduce conflicts
  - Suppose after merging there is shift/reduce conflict on lookahead $a$
    - There is an item $[A \to \alpha \cdot, a]$ in a merged set calling for a reduction by $A \to \alpha$
    - There is another item $[B \to \beta \cdot a\gamma, ?]$ in the set calling for a shift
  - Since the cores of the sets to be merged are the same, there must be a set containing both $[A \to \alpha \cdot, a]$ and $[B \to \beta \cdot a\gamma, ?]$ before merging
  - Then before merging, there is already a shift/reduce conflict on $a$. According to LR(1) parsing table construction algorithm, the grammar is not LR(1). **Contradiction!!!**
- Merging states may cause reduce/reduce conflicts

# Example of Conflicts

- An LR(1) grammar:
  - $S' \to S \qquad S \to aAd \mid bBd \mid aBe \mid bAe \qquad A \to c \qquad B \to c$
- Language: $\{acd, bcd, ace, bce\}$
- One set of valid LR(1) items
  - $\{[A \to c\,\cdot, d], [B \to c\,\cdot, e]\}$
- Another set of valid LR(1) items
  - $\{[B \to c\,\cdot, d], [A \to c\,\cdot, e]$
- After merging, the new item set: $\{[A \to c\,\cdot, d/e], [B \to c\,\cdot, d/e]\}$
  - **Conflict:** reduce $c$ to $A$ or $B$ when the next input symbol is $d/e$?

# Constructing LALR Parsing Table

- Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(1) items
- For each core present among a set of LR(1) items, find all sets having that core, and replace these sets by their union
- Let $C' = \{J_0, J_1, \dots, J_m\}$ be the resulting collection after merging.
  - The parsing actions for state $i$ are constructed from $J_i$ following the LR(1) parsing table construction algorithm.
  - If there is a conflict, this algorithm fails to produce a parser and the grammar is not LALR(1)

**Basic idea:** Merging states in LR(1) parsing table; If there is no reduce-reduce conflict, the grammar is LALR(1), otherwise not LALR(1).

# Constructing LALR Parsing Table

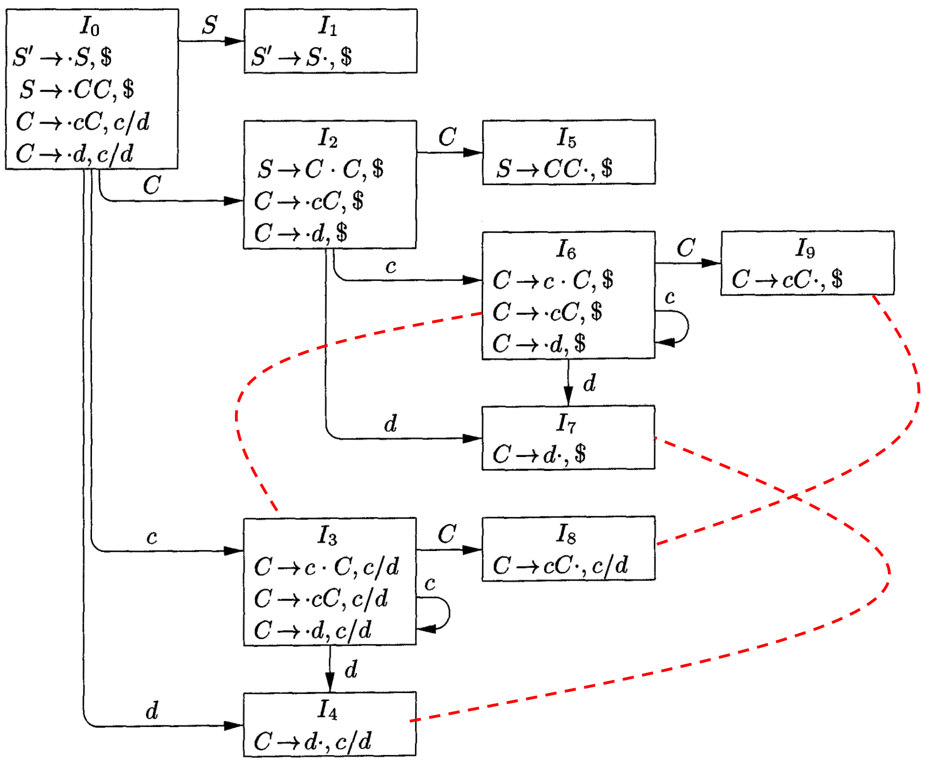
- Construct the GOTO table as follows:
  - If $J$ is the union of one or more sets of LR(1) items, that is $J = I_1 \cup I_2 \cup \cdots \cup I_k$, then the cores of $\mathrm{GOTO}(I_1, X)$, $\mathrm{GOTO}(I_2, X)$, …, $\mathrm{GOTO}(I_k, X)$ are the same, since $I_1, I_2, \dots, I_k$ all have the same core.
  - Let $K$ be the union of all sets of items having the same core as $\mathrm{GOTO}(I_1, X)$
  - $\mathrm{GOTO}(J, X) = K$

Check the previous example to understand the above process:

- $I_3$ and $I_6$ have the same core; $I_{36}$ is the union of the two LR(1) item sets
- GOTO($I_3$, $C$) = $I_8$; GOTO($I_6$, $C$) = $I_9$
- $I_8$ and $I_9$ have the same core; $I_{89}$ is the union of the two LR(1) item sets
- GOTO($I_{36}$, $C$) = $I_{89}$

# LALR Parsing Table Example

- Merging item sets
  - $I_{36}$: $[C \to c \cdot C, c/d/\$], [C \to \cdot cC, c/d/\$], [C \to \cdot d, c/d/\$]$
  - $I_{47}$: $[C \to d \cdot, c/d/\$]$
  - $I_{89}$: $[C \to cC \cdot, c/d/\$]$
- $\text{GOTO}(I_{36}, C) = I_{89}$



# LALR Parsing Table Example

| STATE | ACTION | | | GOTO | |
|---|---|---|---|---|---|
| | $c$ | $d$ | \$ | $S$ | $C$ |
| 0 | s3 | s4 | | 1 | 2 |
| 1 | | | acc | | |
| 2 | s6 | s7 | | | 5 |
| 3 | s3 | s4 | | | 8 |
| 4 | r3 | r3 | | | |
| 5 | | | r1 | | |
| 6 | s6 | s7 | | | 9 |
| 7 | | | r3 | | |
| 8 | r2 | r2 | | | |
| 9 | | | r2 | | |

| STATE | ACTION | | | GOTO | |
|---|---|---|---|---|---|
| | $c$ | $d$ | \$ | $S$ | $C$ |
| 0 | s36 | s47 | | 1 | 2 |
| 1 | | | acc | | |
| 2 | s36 | s47 | | | 5 |
| 36 | s36 | s47 | | | 89 |
| 47 | r3 | r3 | r3 | | |
| 5 | | | r1 | | |
| 89 | r2 | r2 | r2 | | |

# Comparisons Among LR Parsers

- The languages (grammars) that can be handled
  - CLR > LALR > SLR

- # states in the parsing table
  - CLR > LALR = SLR

- Driver programs
  - SLR = CLR = LALR

# Reading Tasks

- Chapter 4 of the dragon book
  - 4.1 Introduction
  - 4.2 Context-Free Grammars
  - 4.3 Writing a Grammar (4.3.1 – 4.3.4)
  - 4.4 Top-Down Parsing (4.4.1 – 4.4.4)
  - 4.5 Bottom-Up Parsing
  - 4.6 Simple LR
  - 4.7 More Powerful LR Parsers (4.7.1 – 4.7.4)
  - 4.8 Using Ambiguous Grammars
  - 4.9 Parser Generators (Lab content)