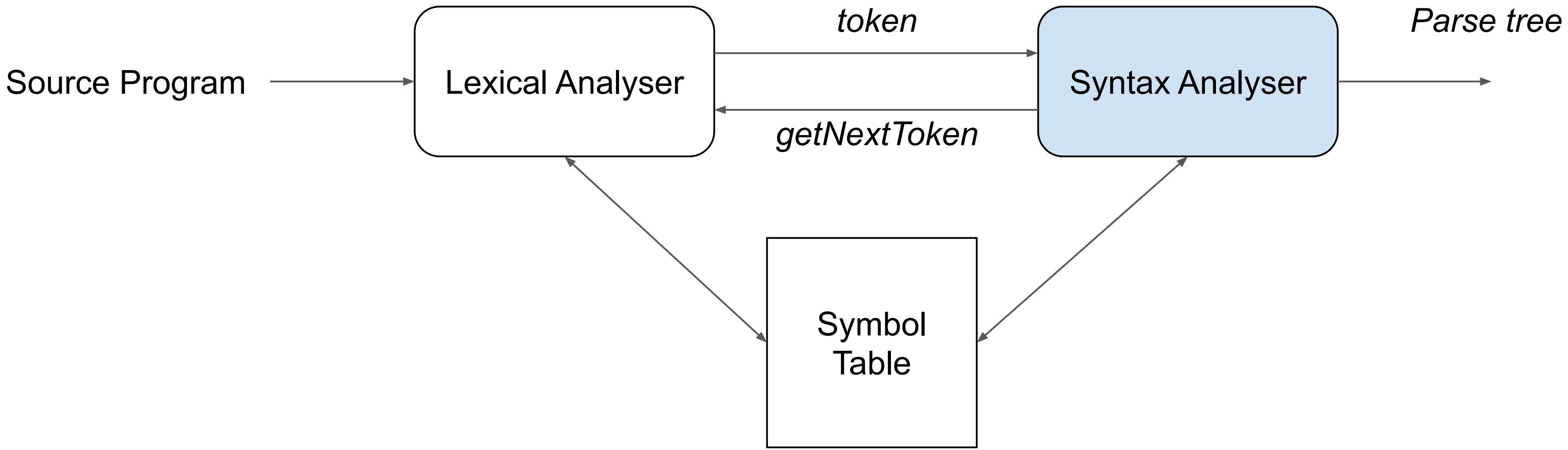
Syntax Analysis

Md Shad Akhtar
Assistant Professor
IIIT Dharwad

Syntax Analyser (Parser)

- Define the syntactic structure for a programming language
- Reads the sequence of tokens from lexical analysis and create|validate the syntactic structure (parse tree) for the sequence of tokens.



Syntactic structure and grammar

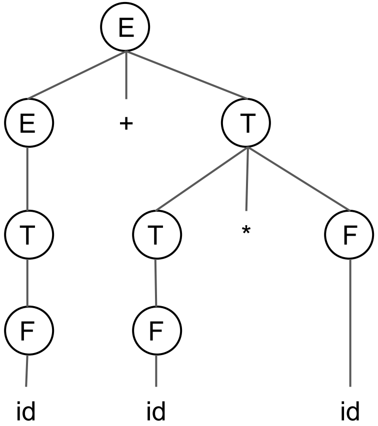
- Syntactic structure is defined by the context-free grammar (CFG)
- Steps to create parse tree
  - Parser checks whether a given source program satisfies the rules implied by a CFG or not
  - If it satisfies, the parser creates the parse tree of that program
  - Otherwise, the parser gives the error messages

Grammar (G)

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid id
\end{aligned}
$$

Token sequence

id + id * id



Syntax Errors

- Role of error handler in parser
  - Report the presence of errors clearly and accurately
  - Recover from each error quickly enough to detect subsequent errors
  - Add minimal overhead to the processing of correct programs.

- Error Detection:
  - Sequence of tokens that can not be accepted by any grammar rule.
  - E.g.:
    - A switch statement without a case statement
    - Missing closing braces
    - Operator without operands c = a +
    - Operands without operator c = a b

Syntax Error Recovery

- Panic-mode:
  - On discovering an error, discards input symbols one at a time until one of a designated set of synchronizing tokens is found, e.g., semicolon, closing brace, etc.
- Phrase-level recovery:
  - Perform local correction on the remaining input to continue
    - Replace the prefix of the remaining input by some string that allows the parser to continue. E.g., Replace comma by semicolon, delete|insert an extra|missing semicolon.
- Error Production
  - For common errors, add special production rules to handle such scenario
- Global correction
  - Ideally, we want as few changes as possible to process incorrect inputs.
  - We can design an algorithm for choosing a minimal sequence of changes to obtain a globally least-cost correction.
    - Given incorrect input x and grammar G, find a correct related input y with as less changes as possible.

Types of parsers

- In general three types of parsers
  - Universal
    - Capable to parse any grammar but too complex to use in compiler
    - E.g.: Cocke-Younger-Kasami (CYK) parser, Earley’s parser
  - Top-down
    - Build parse tree from root to leaf
  - Bottom-Up
    - Build parse tree from leaf to root

Context-free Grammar (CFG)

- Provides a precise syntactic specification of a programming language
- A CFG G = <N, T, P, S>
  - **N**on-terminals:
    - A finite set of non-terminals (variables) [usually in capital letters]
  - **T**erminals:
    - A finite set of terminals (input symbols|tokens) [usually in small letters]
  - **P**roduction:
    - A finite set of productions rules in the following form $A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals (including the empty string); $|A| <= |\alpha|$
  - **S**tart symbol:
    - One of the non-terminal symbols

CFG: An example

- CFG G = <N, T, P, S>
  - **N**on-terminal = {E}
  - **T**erminals = {+, -, *, |, (,), id}
  - **S**tart symbol = {E}
  - **P**roduction

$$E \rightarrow E + E \mid E - E \mid E * E \mid E \mid E \mid E \mid - E$$
$$E \rightarrow (E)$$
$$E \rightarrow id$$

Derivations

- Starting with the start symbol, replace each non-terminals with the body of one of its production rules till all non-terminals are replaced by terminal symbols.
  - $E \Rightarrow E+E \Rightarrow id + E \Rightarrow id + id$
- In general a derivation step is

  $$\alpha A\beta \Rightarrow \alpha\gamma\beta$$

  if there is a production rule $A \rightarrow \gamma$ in our grammar, where α and β are arbitrary strings of terminal and non-terminal symbols.
- $\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)
- $\Rightarrow$ drives in one step
- $\Rightarrow^*$ drives in zero or more steps
- $\Rightarrow^+$ drives in zero or one step

Derivations

- $S \Rightarrow^* \alpha$
  - If α contains non-terminals, it is called as a ***sentential form*** of G
  - If α does not contain non-terminals, it is called as a ***sentence*** of G
- **Left-most derivation:** Always chooses the left-most non-terminal in each derivation step

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$$

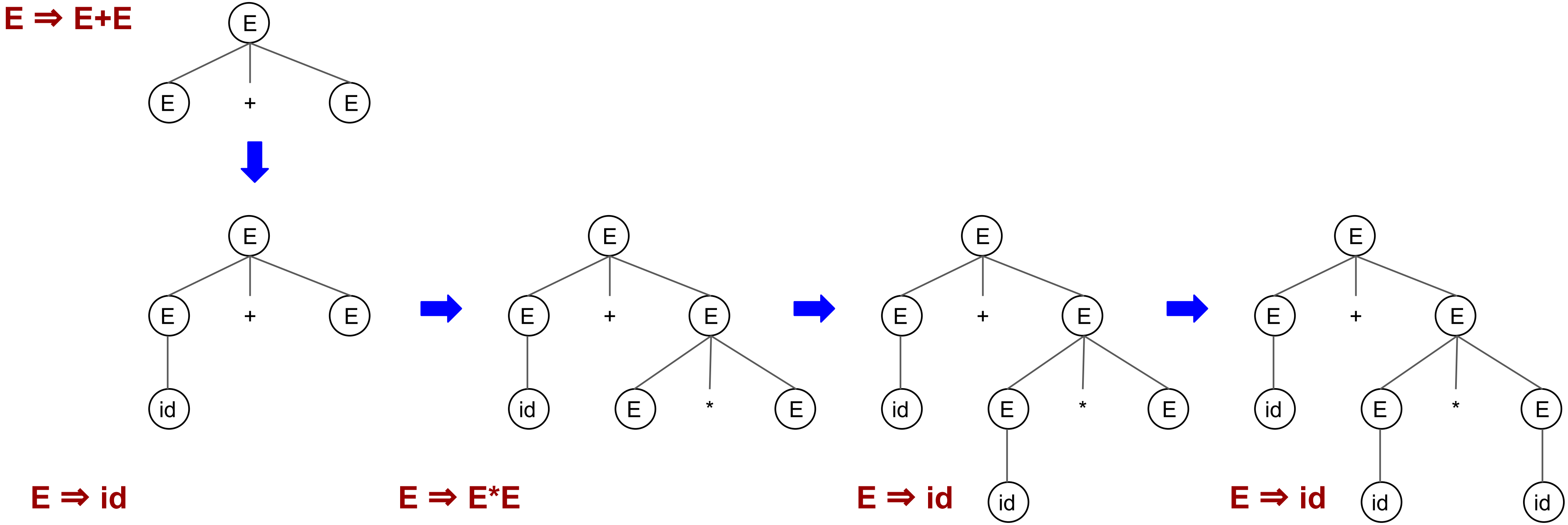
- **Right-most derivation:** Always chooses the right-most non-terminal in each derivation step

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$$

- **Top-down parsers:** Finds the left-most derivation of the given source program
- **Bottom-up parsers:** Finds the right-most derivation of the given source program in the reverse order

Parse Tree

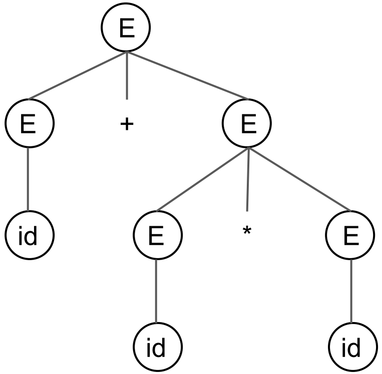
- A graphical representation of a derivation
- Intermediate nodes: Inner nodes of a parse tree
- Leaves: Terminal symbols



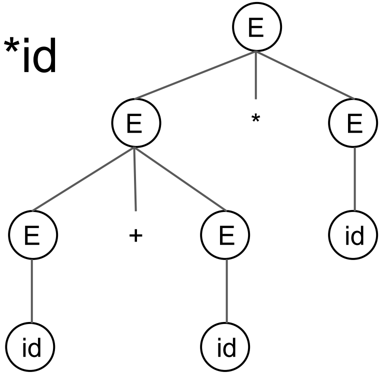
Ambiguity

- A grammar that produces more than one parse tree for a sentence is called as an ambiguous grammar

- E ⇒ E+E ⇒ id+E ⇒ id+E*E ⇒* id+id*E ⇒ id+id*id



- E ⇒ E*E ⇒ E+E*E ⇒ id+E*E ⇒ id+id*E ⇒ id+id*id



Ambiguity and Parser

- For the most parsers, the grammar must be unambiguous.
  - unique selection of the parse tree for a sentence

- Disambiguation of an ambiguous grammar
  - Necessary to eliminate the ambiguity in the grammar during the design phase of the compiler
  - Choose one of the parse trees of a sentence to restrict to this choice

Ambiguity disambiguation

- Stmt → if Expr then Stmt | if Expr then Stmt else Stmt | other_stmts
- Input string: if E_1 then if E_2 then S_1 else S_2
- **Interpretation 1:** S_2 being executed when E_1 is false (thus attaching the else to the first if)
    - if E_1 then (if E_2 then S_1) else S_2
- **Interpretation 2:** S_2 being executed when E_1 is true and E_2 is false (thus attaching the else to the second if)
    - if E_1 then (if E_2 then S_1 else S_2)

Ambiguity disambiguation

- In general, we prefer the second parse tree (else matches with closest if)
- So, we have to disambiguate our grammar to reflect this choice
- Unambiguous grammar:

Stmt	→	matchedStmt | unmatchedStmt
matchedStmt	→	if Expr then matchedStmt else matchedStmt |
		Otherstmts
unmatchedStmt	→	if Expr then Stmt |
		if Expr then matchedStmt else unmatchedStmt

Ambiguity disambiguation

- Operator precedence grammar:

$$E \rightarrow E{+}E \mid E{*}E \mid E\hat{\ }E \mid id \mid (E)$$

- Unambiguous grammar

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

Precedence

^ (right to left)
* (left to right)
+ (left to right)

Left Recursion

- A grammar is left recursive if it has a non-terminal A such that there is a derivation
    - $A \Rightarrow^{+} A\alpha$ for some string α

- Top-down parsing techniques ***cannot handle*** left-recursive grammars
    - Conversion of left-recursive grammar into an equivalent non-recursive grammar is ***mandatory***.
- Possible ways of left-recursion
    - It may appear in a single step of the derivation (immediate left-recursion)
    - It may appear in more than one step of the derivation

Removing Left Recursion

In general,

$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ Where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$$A \rightarrow \beta_1 A' \mid \ldots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

an equivalent grammar

Removing Left Recursion: An example

$$
\begin{aligned}
E &\rightarrow E+T \mid T \\
T &\rightarrow T*F \mid F \\
F &\rightarrow id \mid (E)
\end{aligned}
$$

$\Downarrow$ eliminate immediate left recursion

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *F\ T' \mid \varepsilon \\
F &\rightarrow id \mid (E)
\end{aligned}
$$

Why left-recursion is a problem?

- Given
    - `A → Aa | b`

  generate a top-down parse tree from input string ‘aaaaaa’

- On first input symbol ‘a’, you appy first production since second production expect first character to be ‘b’. [Note that you don’t know what’s your second input]
    - `A ⇒ Aa ⇒ Aaa ⇒        ⇒ Aaaaaaa`
    - We are waiting to reduce ‘A’ to ‘a’
    - After infinite/many steps, we may get to know that the path we chose was not correct.

Non-immediate Left-recursion

- A grammar cannot be immediately left-recursive, but it still can be left-recursive
- Just elimination of the immediate left-recursion does not guarantee a grammar which is not left-recursive

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Sc \mid d$$

- This grammar is not immediately left-recursive, but it is still left-recursive

$$S \Rightarrow Aa \Rightarrow Sca$$

Or

$$A \Rightarrow Sc \Rightarrow Aac$$

Elimination of left-recursion: Algorithm

Input: A grammar G without *e-moves* or *cycle*

Output: An equivalent grammar without left recursion

1. Arrange non-terminals in some order: $A_1 \dots A_n$
2. for i = 1 to n
   a. for j = 1 to i-1
      i. replace each production of the form

$$A_i \rightarrow A_j\gamma \qquad \Rightarrow \qquad \begin{aligned} &A_i \rightarrow \alpha_1\gamma \mid \alpha_2\gamma \mid \dots \mid \alpha_k\gamma, \\ &A_j \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k \end{aligned}$$

   b. eliminate the immediate left-recursions among A_i productions

If there are e-moves, the algorithm does not guarantee to work.

Elimination of left-recursion: Example

- Let grammar G: $S \rightarrow Aa \mid b$
  $A \rightarrow Ac \mid Sd \mid f$
- Order of non-terminals: S, A
- For S: There is no immediate left recursion in S.
- For A: Replace $A \rightarrow Sd$ with $A \rightarrow Aad \mid bd \Rightarrow A \rightarrow Ac \mid Aad \mid bd \mid f$

  Eliminate the immediate left-recursion in A

  $A \rightarrow bdA' \mid fA'$

  $A' \rightarrow cA' \mid adA' \mid \varepsilon$
- So, the resulting equivalent grammar which is not left-recursive is:

  $S \rightarrow Aa \mid b$

  $A \rightarrow bdA' \mid fA'$

  $A' \rightarrow cA' \mid adA' \mid \varepsilon$

Elimination of left-recursion: Exercises

1. $A \to ABd \mid Aa \mid a$

   $B \to Be \mid b$

2. $A \to Ba \mid Aa \mid c$

   $B \to Bb \mid Ab \mid d$

3. $X \to XSb \mid Sa \mid b$

   $S \to Sb \mid Xa \mid a$

Elimination of left-recursion: Solutions

1. $A \to aA'$

   $A' \to BdA' \mid aA' \mid \varepsilon$

   $B \to bB'$

   $B' \to eB' \mid \varepsilon$

2. $A \to BaA' \mid cA'$

   $A' \to aA' \mid \varepsilon$

   $B \to cA'bB' \mid dB'$

   $B' \to bB' \mid aA'bB' \mid \varepsilon$

3. $X \to SaX' \mid bX'$

   $X' \to SbX' \mid \varepsilon$

   $S \to bX'aS' \mid aS'$

   $S' \to bS' \mid aX'aS' \mid \varepsilon$

Left-factoring

- Top-down parser without backtracking (predictive parser) insists that the grammar must be left left-factored

```
stmt   →   if expr then stmt else stmt |
           if expr then stmt
```

- After seeing `if`, we cannot decide which production rule to choose to re-write stmt in the derivation

Left-factoring

- In general,

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

  where α is non-empty and the first symbols of β_1 and β_2 (if they have one) are different
- Choice involved when processing α

$$A \text{ to } \alpha\beta_1 \text{ or}$$
$$A \text{ to } \alpha\beta_2$$

- Rewrite the grammar as follows:

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

  so, we can immediately expand $A \rightarrow \alpha A'$

Elimination of Left-factoring: Algorithm

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix,

$$A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$$

Convert it into

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$$
$$A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$$

Elimination of Left-factoring: Example

Example 1:

$A \rightarrow abB \mid aB \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid cdg \mid cdeB \mid cdfB$
$A' \rightarrow bB \mid B$

$\Downarrow$

$A \rightarrow aA' \mid cdA''$
$A' \rightarrow bB \mid B$
$A'' \rightarrow g \mid eB \mid fB$

Example 2:

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$\Downarrow$

$A \rightarrow aA' \mid b$
$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

$\Downarrow$

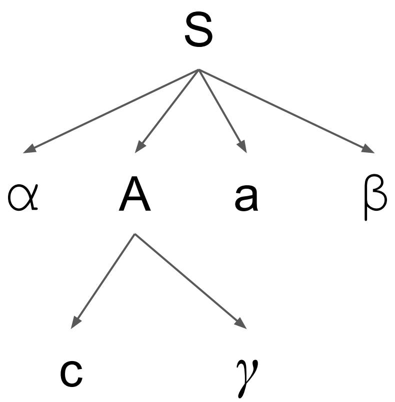
$A \rightarrow aA' \mid b$
$A' \rightarrow d \mid \varepsilon \mid bA''$
$A'' \rightarrow \varepsilon \mid c$

FIRST() and FOLLOW()

Lookahead symbols

- The construction of top-down and bottom-up parsing is aided by two functions on grammar G
  - FIRST(α): The set of *first character* that can be derived from α
  - FOLLOW(A): The set of *character that can come immediately after* the non-terminal A.

$$S \rightarrow \alpha \ A \ a \ \beta$$
$$A \rightarrow c \ \gamma$$



FIRST(a) = {a}
FIRST(A) = FIRST(c) = {c}
FIRST(S) = FIRST(α) = {...}
FIRST(β) = {...}
FIRST(γ) = {...}

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

$: A special symbol for the end marker.

FIRST()

- FIRST(α)
  a. If α is a terminal
     - FIRST(α) = {α}

  b. If α is a non-terminal and $\alpha \longrightarrow \beta_1\ \beta_2\ \beta_3 \ldots \beta_k$
     - FIRST(α) = FIRST(α) U FIRST(β_i) if $\beta_1\ \beta_2 \ldots \beta_{i-1} \Rightarrow^* \varepsilon$

  c. If $\alpha \longrightarrow \varepsilon$
     - FIRST(α) = FIRST(α) U ε

FOLLOW()

- FOLLOW(A)
  a. If A is the start symbol and \$ is the special end marker
     - FOLLOW(A) $=$ {\$}

  b. If $A \rightarrow \alpha B \beta$
     - FOLLOW(B) $=$ FOLLOW(B) $\cup$ { FIRST(β) - ε }

  c. If $A \rightarrow \alpha B$ OR $A \rightarrow \alpha B \beta$ with FIRST(β) has ε
     - FOLLOW(B) $=$ FOLLOW(B) $\cup$ FOLLOW(A)

FIRST() and FOLLOW()

1. G: $A \rightarrow aBe \mid cBd \mid C$
   $B \rightarrow bB \mid \varepsilon$
   $C \rightarrow f$

FIRST(a) = {a}
FIRST(b) = {b}
FIRST(c) = {c}
FIRST(d) = {d}
FIRST(e) = {e}
FIRST(f) = {f}
FIRST(A) = {a, c, f}
FIRST(B) = {b, ε}
FIRST(C) = {f}

FOLLOW(A) = {$}
FOLLOW(B) = {e,d}
FOLLOW(C) = {$}

2. G: $A \rightarrow aBc$
   $B \rightarrow bC$
   $C \rightarrow c \mid \varepsilon$

FIRST(a) = {a}
FIRST(b) = {b}
FIRST(c) = {c}
FIRST(A) = {a}
FIRST(B) = {b}
FIRST(C) = {c, ε}
FOLLOW(A) = {$}
FOLLOW(B) = {c}
FOLLOW(C) = {c}

FIRST() and FOLLOW()

3. G: E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → id | (E)

FIRST(+) = {+}, FIRST(*) = {*}, FIRST(id) = {id}, FIRST('(') = { (}, FIRST(')') = {) }
FIRST(E) = FIRST(T) = FIRST(F) = { id, (}
FIRST(T') = {*, ε}
FIRST(E') = {+, ε}

FOLLOW(E) = FOLLOW(E') = {), $}
FOLLOW(T) = FOLLOW(T') = {+,), $}
FOLLOW(F) = {+, *,), $}

Top-Down Parsing

Top-Down Parsing

- Parse tree are created top to bottom
  - Begin with the start symbol to generate the input string.

- Top-down parser
  - Recursive-Descent Parsing
  - Predictive Parsing
  - Non-recursive Predictive Parsing (LL(1) parsing)

Recursive-Descent Parsing

- Tries to find the left-most derivation
- Recursively applies production rules
- If current production fails, ***backtrack***, apply another rule.
- Its simple but not widely used
- Not efficient
  - Cost of backtracking is involved which may be huge.

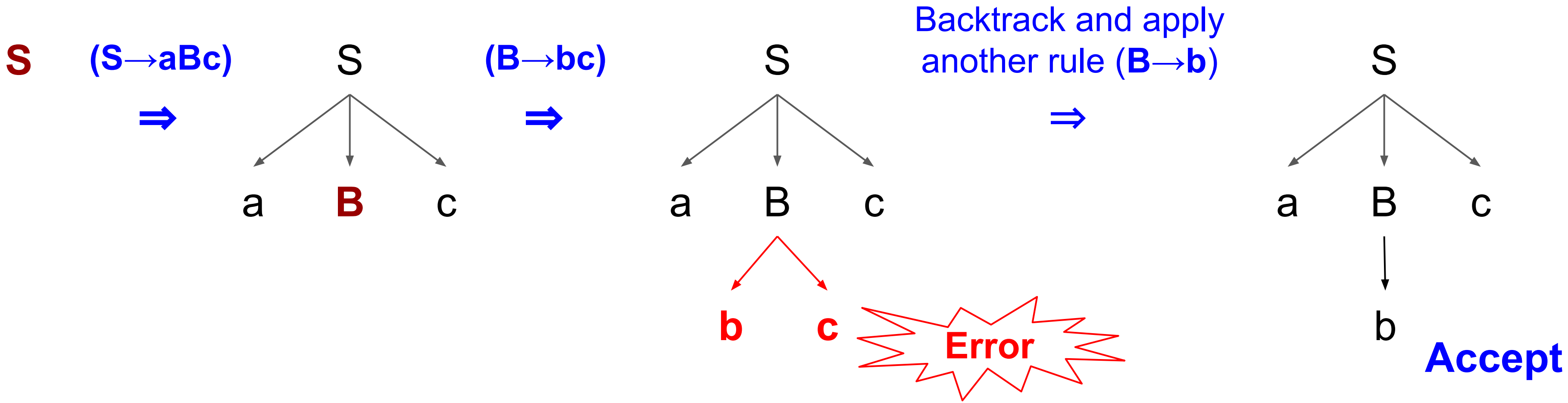
Recursive-Descent Parsing

- Let grammar G:

  $S \rightarrow aBc$

  $B \rightarrow bc \mid b$

- Input string: **a b c**



Designing a recursive-descent parser

- Write a procedure/function for each non-terminal.
- Call the associated function whenever a non-terminal is encountered during derivation.

```
function S()
{
   // match the input and/or call non-terminal functions
   // Backtrack, if it does not apply.
}
```

Designing a recursive-descent parser

- Design a recursive-descent parser for the following grammar.

$$S \rightarrow aBc$$

$$B \rightarrow bc \mid b$$

(Recursive) Predictive Parser

- A special form of recursive-descent parsing without backtracking.
- Since no backtracking, its efficient
- But needs a special kind of grammar, i.e., LL(1) grammar
- Uniquely choose a production rule by looking at the current symbol in the input string

Production $\qquad A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$

Input: $\qquad \ldots\ a\ \ldots$

Current token

Predictive Parser

- **Constraints on grammar**
    a. Unambiguous
    b. No left recursion should be there
    c. Grammar should be left-factored
- Still, no 100% guarantee

Predictive Parser

- Let grammar G:

$$A \rightarrow aBe \mid cBd \mid C$$
$$B \rightarrow bB \mid \varepsilon$$
$$C \rightarrow f$$

- Left recursion?
  - No left recursion ⇒ This ensures that, given the current token, we don’t have to backtrack
- Left factored?
  - Yes ⇒ This ensures that, for a input symbol, we no longer have to make a decision

Predictive Parser

- For predictive parser, we need **lookahead symbols**
  a. Compute **FIRST()** and **FOLLOW()** for the grammar

- For a given non-terminal A and the current input symbol a
  a. IF FIRST(A) contains symbol a,
     - Apply the production associated with symbol a.
  b. ElseIF FIRST(A) contains symbol ε,
     - IF FOLLOW(A) contains symbol a,
       - Apply the production $A \rightarrow$ ε and proceed.
  c. Else
     - Error

Input string: **a b e**

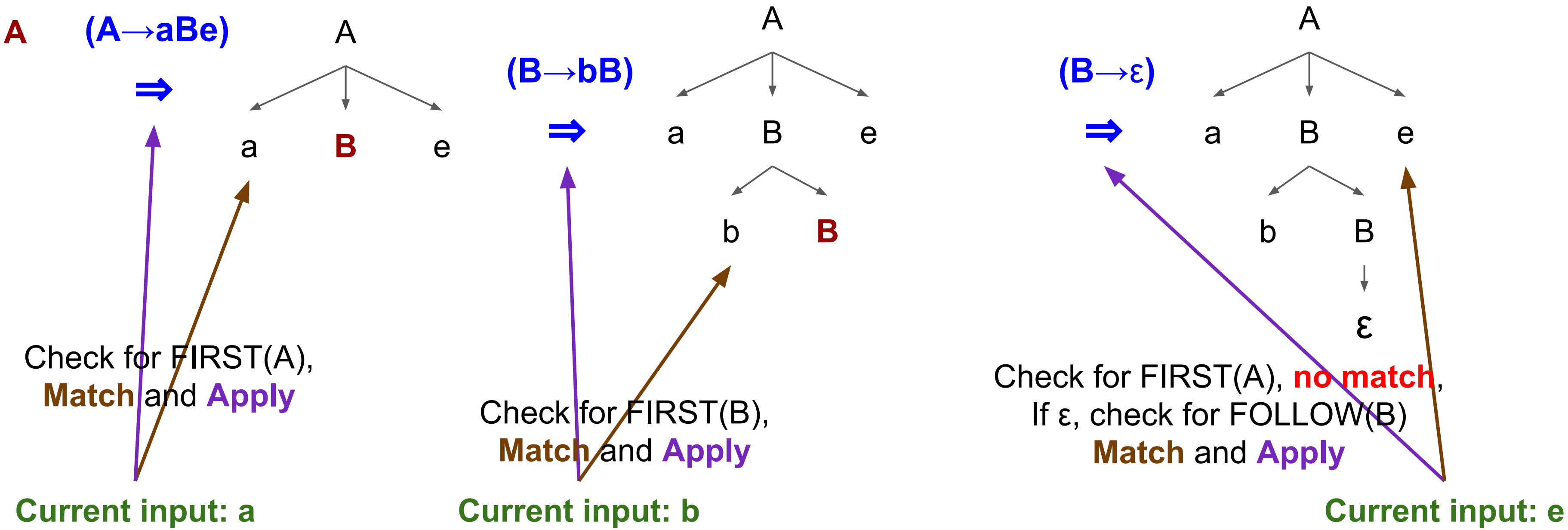
Predictive Parser

- Grammar G:

  A → aBe | cBd | C
  B → bB | ε
  C → f

FIRST(A) = {a, c, f}
FIRST(B) = {b, ε}
FIRST(C) = {f}

FOLLOW(A) = {\$}
FOLLOW(B) = {e, d}
FOLLOW(C) = {\$}



Predictive Parser

- Let grammar G:

$$S \rightarrow aBc$$
$$B \rightarrow bc \mid b$$

- Left recursion?
  - No left recursion

- Left factored?
  - No

$$S \rightarrow aBc$$
$$B \rightarrow bB'$$
$$B' \rightarrow c \mid \varepsilon$$

Predictive Parser

- Let the new grammar G':

  $S \rightarrow aBc$

  $B \rightarrow bB'$

  $B' \rightarrow c \mid \varepsilon$

- Find FIRST and FOLLOW

  a. FIRST(S) = {a} FIRST(B) = {b} FIRST(B') = {c, ε}

  b. FOLLOW(S) = {$} FOLLOW(B) = {c} FOLLOW(B') = {c}

- Input string: **a b c**

Designing a Predictive Parser

- Write a procedure/function for each non-terminal.
- Call the associated function whenever a non-terminal is encountered during derivation.
- Match lookahead with the current input and apply the rule

```
function S(lookahead, current)
{
    // match the input and/or call non-terminal functions
}
```

Designing a Predictive Parser

- Design a recursive-descent parser for the following grammar.

$$A \rightarrow aBe \mid cBd \mid C$$

$$B \rightarrow bB \mid \varepsilon$$

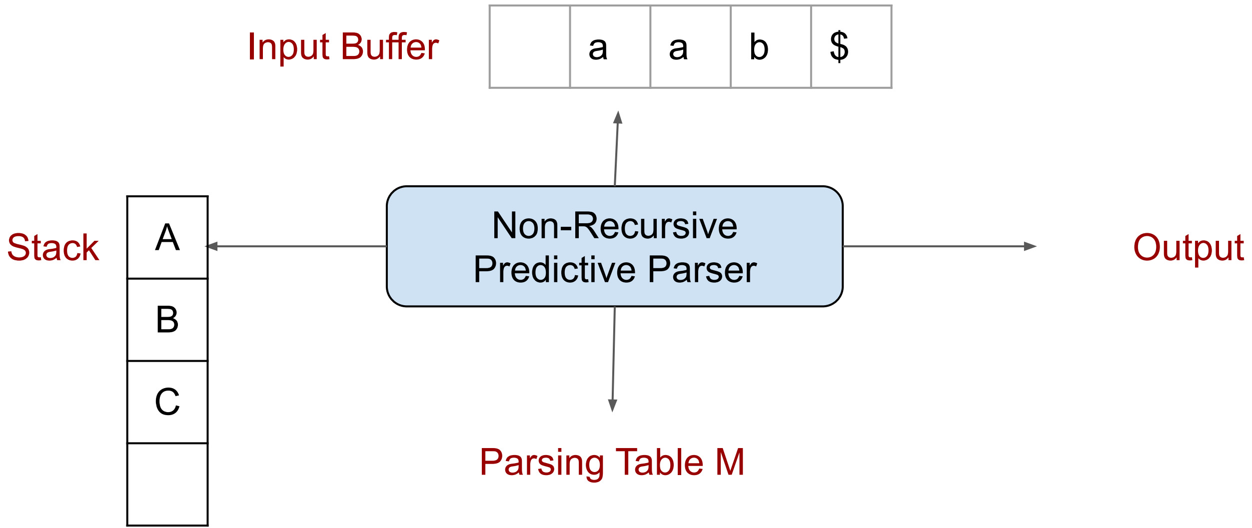
$$C \rightarrow f$$

```
proc A {
    current token {
        a:
            - match the current token with a, and move to the next token;
            - call B;
            - match the current token with e, and move to the next token;
        c:
            - match the current token with c, and move to the next token;
            - call B;
            - match the current token with d, and move to the next token;
        f:
            - call C
    }}
proc B {
    current token {
        b:
            - match the current token with b, and move to the next token;
            - call B
        e,d:
            do nothing              //FOLLOW(B)
    }}
proc C {
        f:  - match the current token with f, and move to the next token;
    }
```

LL(1) Parser

Non-Recursive Predictive or LL(1) Parser

- Top-down parser
- Table-driven parser
- LL(k), with k = 1
  - Left-to-right Left-most-derivation with k lookahead symbols



Non-Recursive Predictive or LL(1) Parser

- **Input buffer**
  - Contains the string to be parsed with end marked with a special symbol $
- **Output**
  - A production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer
- **Stack**
  - Contains the grammar symbols
  - At the bottom of the stack, there is a special end marker symbol $
  - Initially the stack contains only the symbol $ and the starting symbol S
    - $S ← Initial stack
  - Parsing completes when both input and stack becomes empty (i.e., only $ left in stack)
- **Parsing table**
  - A two-dimensional array M[A, a]
  - Each row is a non-terminal symbol
  - Each column is a terminal symbol or the special symbol $
  - Entries holds a production rule.

LL(1) Grammar

- For any grammar G, if we can build an LL(1), then the grammar is called LL(1) grammar.
  - No *left-recursive*, *non-left-factored* or *ambiguous* grammar can be LL(1)
  - Still, there are some grammar which are *non-left-recursive*, *left-factored* and *unambiguous* but not a LL(1) grammar.

- A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:
  - Both α and β cannot derive strings starting with same terminals
  - At most one of α and β can derive to ε
  - If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A) and vice-versa.

Construction of LL(1) parsing table

Input: Grammar G.

Output: Parsing Table M.

1. For each production $A \rightarrow \alpha$ of the grammar,
2. do
   a. For each terminal a in FIRST(α)
      i. Add $A \rightarrow \alpha$ to M[A, a]
   b. If ε is in FIRST(α)
      i. For each terminal a in FOLLOW(A)
         1. Add $A \rightarrow \alpha$ to M[A, a]
   c. If ε is in FIRST(α) and \$ is in FOLLOW(A)
      i. Add $A \rightarrow \alpha$ to M[A, a]

Construction of LL(1) parsing table

$E \rightarrow TE'$ $\quad E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$ $\quad T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow id \mid (E)$

FIRST(T') = {*, ε} FIRST(E) = FIRST(T) = FIRST(F) = { id, (}
FIRST(E') = {+, ε} FOLLOW(E) = FOLLOW(E') = {), $}
FOLLOW(F) = {+, *,), $} FOLLOW(T) = FOLLOW(T') = {+,), $}

Non-Term	**Input Symbols**					
	id	+	*	(	)	$
E						
E'						
T						
T'						
F						

Construction of LL(1) parsing table

$E \rightarrow TE'$	$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$	$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow id \mid (E)$	

FIRST(T') = {*, ε}	FIRST(E) = FIRST(T) = FIRST(F) = { id, (}
FIRST(E') = {+, ε}	FOLLOW(E) = FOLLOW(E') = {), $}
FOLLOW(F) = {+, *,), $}	FOLLOW(T) = FOLLOW(T') = {+,), $}

Production $E \rightarrow TE'$ ⇒ First(TE') = First(T) = { (, id } ⇒ Add $E \rightarrow TE'$ to M[E, id] and M[E, (]

Non-Term	**Input Symbols**					
	id	+	*	(	)	$
E	***E → TE'***			***E → TE'***		
E'						
T						
T'						
F						

Construction of LL(1) parsing table

$E \rightarrow TE'$	$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$	$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow id \mid (E)$	

FIRST(T') = {*, ε}	FIRST(E) = FIRST(T) = FIRST(F) = { id, (}
FIRST(E') = {+, ε}	FOLLOW(E) = FOLLOW(E') = {), $}
FOLLOW(F) = {+, *,), $}	FOLLOW(T) = FOLLOW(T') = {+,), $}

Production $T \rightarrow FT'$ ⇒ First(FT') = First(F) = { (, id } ⇒ Add $T \rightarrow FT'$ to M[T, id] and M[T, (]

Non-Term	Input Symbols					
	id	+	*	(	)	$
E	*E → TE'*			*E → TE'*		
E'						
T	***T → FT'***			***T → FT'***		
T'						
F						

Construction of LL(1) parsing table

$E \to TE'$ $\quad$ $E' \to +TE' \mid \varepsilon$
$T \to FT'$ $\quad$ $T' \to *FT' \mid \varepsilon$
$F \to id \mid (E)$

FIRST(T') = {*, ε} $\quad$ FIRST(E) = FIRST(T) = FIRST(F) = { id, (}
FIRST(E') = {+, ε} $\quad$ FOLLOW(E) = FOLLOW(E') = {), $}
FOLLOW(F) = {+, *,), $} $\quad$ FOLLOW(T) = FOLLOW(T') = {+,), $}

Production $F \to id$ ⇒ First(id) = { id } ⇒ Add $F \to id$ to M[F, id]

Non-Term	Input Symbols					
	id	+	*	(	)	$
E	*E → TE'*			*E → TE'*		
E'						
T	*T → FT'*			*T → FT'*		
T'						
F	***F → id***					

Construction of LL(1) parsing table

$E \rightarrow TE'$	$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$	$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow id \mid (E)$	

FIRST(T') = {*, ε}   FIRST(E) = FIRST(T) = FIRST(F) = { id, (}
FIRST(E') = {+, ε}   FOLLOW(E) = FOLLOW(E') = {), $}
FOLLOW(F) = {+, *,), $}   FOLLOW(T) = FOLLOW(T') = {+,), $}

Production $F \rightarrow (E)$ ⇒ First((E)) = { (} ⇒ Add $F \rightarrow (E)$ to M[F, (]

Non-Term	**Input Symbols**					
	id	+	*	(	)	$
E	*E → TE'*			*E → TE'*		
E'						
T	*T → FT'*			*T → FT'*		
T'						
F	*F → id*			***F → (E)***		

Construction of LL(1) parsing table

$E \rightarrow TE'$	$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$	$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow id \mid (E)$	

FIRST(T') = {*, ε}
FIRST(E') = {+, ε}
FOLLOW(F) = {+, *,), $}
FIRST(E) = FIRST(T) = FIRST(F) = { id, (}
FOLLOW(E) = FOLLOW(E') = {), $}
FOLLOW(T) = FOLLOW(T') = {+,), $}

Production $E' \rightarrow +TE'$ ⇒ First(+TE') = { + } ⇒ Add $E' \rightarrow +TE'$ to M[E', +]

Non-Term	**Input Symbols**					
	id	+	*	(	)	$
E	*E → TE'*			*E → TE'*		
E'		***E' → +TE'***				
T	*T → FT'*			*T → FT'*		
T'						
F	*F → id*			*F → (E)*		

Construction of LL(1) parsing table

E → TE'	E' → +TE' | ε
T → FT'	T' → *FT' | ε
F → id | (E)	

FIRST(T') = {*, ε} FIRST(E) = FIRST(T) = FIRST(F) = { id, (}
FIRST(E') = {+, ε} FOLLOW(E) = FOLLOW(E') = {), $}
FOLLOW(F) = {+, *,), $} FOLLOW(T) = FOLLOW(T') = {+,), $}

Production T' → *FT' ⇒ First(*FT') = { * } ⇒ Add T' → *FT' to M[T', *]

Non-Term	Input Symbols					
	id	+	*	(	)	$
E	*E → TE'*			*E → TE'*		
E'		*E' → +TE'*				
T	*T → FT'*			*T → FT'*		
T'			***T' → *FT'***			
F	*F → id*			*F → (E)*		

Construction of LL(1) parsing table

$E \rightarrow TE'$ $\quad$ $E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$ $\quad$ $T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow id \mid (E)$

FIRST(T') = {*, ε} $\quad$ FIRST(E) = FIRST(T) = FIRST(F) = { id, (}
FIRST(E') = {+, ε} $\quad$ FOLLOW(E) = FOLLOW(E') = {), $}
FOLLOW(F) = {+, *,), $} $\quad$ FOLLOW(T) = FOLLOW(T') = {+,), $}

Production $E' \rightarrow \varepsilon$ ⇒ Follow(E') = {), $ } ⇒ Add $E' \rightarrow \varepsilon$ to M[E',)] and M[E', $]

Non-Term	Input Symbols					
	id	+	*	(	)	$
E	*E → TE'*			*E → TE'*		
E'		*E' → +TE'*			***E' → ε***	***E' → ε***
T	*T → FT'*			*T → FT'*		
T'			*T' → *FT'*			
F	*F → id*			*F → (E)*		

Construction of LL(1) parsing table

$E \rightarrow TE'$	$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$	$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow id \mid (E)$	

FIRST(T') = {*, ε}
FIRST(E) = FIRST(T) = FIRST(F) = { id, (}
FIRST(E') = {+, ε}
FOLLOW(E) = FOLLOW(E') = {), $}
FOLLOW(F) = {+, *,), $}
FOLLOW(T) = FOLLOW(T') = {+,), $}

Production $T' \rightarrow \varepsilon$ ⇒ Follow(T') = {+,), $ } ⇒ Add $T' \rightarrow \varepsilon$ to M[T', +], M[T',)] and M[T', $]

Non-Term	Input Symbols					
	id	+	*	(	)	$
E	*E → TE'*			*E → TE'*		
E'		*E' → +TE'*			*E' → ε*	*E' → ε*
T	*T → FT'*			*T → FT'*		
T'		***T' → ε***	*T' → *FT'*		***T' → ε***	***T' → ε***
F	*F → id*			*F → (E)*		

LL(1) parsing

- Once we have built a parsing table M, verify whether a given string w is part of the language or not.
- **LL(1) parsing algorithm**
  - Look at the symbol at the top of the stack (e.g., X) and the current symbol in the input string (e.g., a)
    - If X == a == \$ → Halt with success;
    - If X == a != \$ → Pop; Move to next input;
    - If X == terminal OR M[X, a] is empty → Halt with error;
    - If X in non-terminal and M[X, a] == X → $\alpha_1\alpha_2\alpha_3 \ldots \alpha_k$
      - Pop
      - Push $\alpha_k\alpha_{k-1}\alpha_{k-2} \ldots \alpha_1$ [top of stack = α_1]
      - Output the production X → $\alpha_1\alpha_2\alpha_3 \ldots \alpha_k$

LL(1) parsing

Input: id + id * id

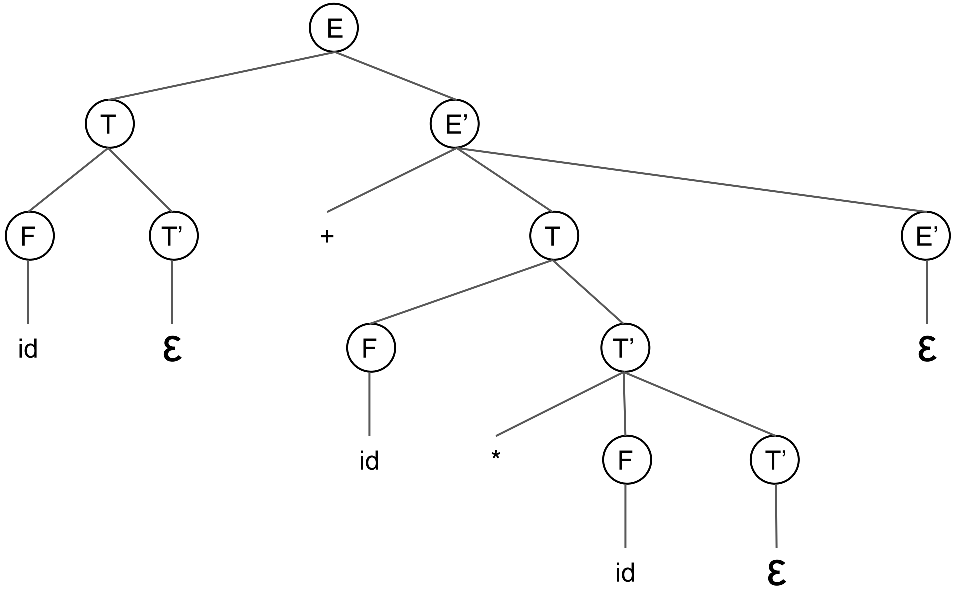
Matched	STACK	Input	Output
	E $	id + id * id $	
	T E' $	id + id * id $	Output: E → T E'
	F T' E' $	id + id * id $	Output: T → F T'
	id T' E' $	**id** + id * id $	Output: F → id
id	T' E' $	+ id * id $	Match: **id**
id	E' $	+ id * id $	Output: T' → ε
id	**+** T E' $	**+** id * id $	Output: E' → + T E'
id +	T E' $	id * id $	Match: **+**
id +	F T' E' $	id * id $	Output: T → F T'
id +	**id** T' E' $	**id** * id $	Output: F → id
id + id	T' E' $	* id $	Match: **id**
id + id	***** F T' E' $	***** id $	Output: T' → * F T'
id + id *	F T' E' $	id $	Match: *****
id + id *	**id** T' E' $	**id** $	Output: F → id
id + id * id	T' E' $	$	Match: **id**
id + id * id	E' $	$	Output: T' → ε
id + id * id	$	$	Output: E' → ε

Stack: Empty
Input: Empty
Accept and stop!!

LL(1) parsing

- Following the output sequence gives you left-most derivation for the input

E → T E’
→ F T’ E’
→ id T’ E’
→ id ε E’
→ id + T E’
→ id + F T’ E’
→ id + id T’ E’
→ id + id * F T’ E’
→ id + id * id T’ E’
→ id + id * id ε E’
→ id + id * id ε



LL(1) parsing

- Construct parsing table for the following grammar:

  $S \rightarrow i\ E\ t\ S\ \ |\ i\ E\ t\ S\ e\ S\ |\ a$

  $E \rightarrow b$

[S' on e] = ??

Non-Term	Input Symbols					
	a	b	e	i	t	$
S	***S → a***			***S → i E t S S'***		
S'			***S' → ε*** ***S' → eS***			***S' → ε***
E		***E → b***				

Error Recovery in LL(1) Parsing

- An error may occur in the predictive parsing (LL(1) parsing), if
  - The terminal symbol on the top of stack does not match with the current input symbol
  - top of stack is a non-terminal A, the current input symbol is a, and the parsing table entry M[A, a] is empty.

Panic-mode Error Recovery in LL(1) Parsing

- Skip over the symbols on the input until a synchronization [sync] token is found.
- Synchronization tokens
  - Place all the symbols in the FOLLOW(A) into the synchronizing token set for the non-terminal A.

- If a non-terminal A can generate ε, then A → ε can be used as default choice.
- If a terminal on the top of stack cannot be matched, pop the terminal.
- If a non-terminal on the top of stack has an entry sync on a terminal a, skip the terminal.

Modified LL(1) parsing table with “sync” tokens

$E \rightarrow TE'$ $E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$ $T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow id \mid (E)$

FIRST(T’) = {*, ε} FIRST(E) = FIRST(T) = FIRST(F) = { id, (}
FIRST(E’) = {+, ε} FOLLOW(E) = FOLLOW(E’) = {), $}
FOLLOW(F) = {+, *,), $} FOLLOW(T) = FOLLOW(T’) = {+,), $}

Non-Term	Input Symbols					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	**sync**	**sync**
E’		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$	**sync**		$T \rightarrow FT'$	**sync**	**sync**
T’		$\boldsymbol{T' \rightarrow \varepsilon}$	$T' \rightarrow *FT'$		$\boldsymbol{T' \rightarrow \varepsilon}$	$\boldsymbol{T' \rightarrow \varepsilon}$
F	$F \rightarrow id$	**sync**	**sync**	$F \rightarrow (E)$	**sync**	**sync**

Panic-mode Recovery in LL(1) parsing

Input:) id * + id

STACK	Input	Remarks
E $	**) id * + id $**	**Error, M[E,)] = sync, skip)**
E $	id * + id $	id is in FIRST(E)
T E’ $	id * + id $	
F T’ E’ $	id * + id $	
id T’ E’ $	**id** * + id $	
T’ E’ $	* + id $	
* F T’ E’ $	* + id $	
F T’ E’ $	**+ id $**	**Error, M[F, +] = sync, skip +**
F T E’ $	id $	id is in FIRST(F)
id T’ E’ $	**id** $	
T’ E’ $	$	
E’ $	$	
$	$	

Bottom-Up Parsing

Bottom-Up parsing

- Construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top)
  - Reducing a string w to the start symbol of a grammar.
  - At each reduction step a particular substring matching the right side of a production is replaced by the symbol on the left of that production.
  - Gives the right-most derivation in the reverse order.

Bottom-Up parsing: An example

G:	$S \rightarrow aABe$
	$A \rightarrow Abc \mid b$
	$B \rightarrow d$
Input:	abbcde

- Procedure
  - Scan the string from left to right looking for a substring that matches the right side of a production: b and d qualifies
    - Choose *leftmost* b and apply A→ b, So string becomes aAbcde
  - Scan left to right: Abc, b and d qualifies
    - Choose *leftmost* Abc and apply A→ Abc, So string becomes aAde
  - Scan left to right: d qualifies
    - Apply B → d, so the string becomes aABe
  - Scan left to right: aABe qualifies
    - Apply S → aABe
- abbcde ⇒ aAbcde ⇒ aAde ⇒ aABe ⇒ S

Right-most derivation

Handle

- Handle of a string is a substring that
  - *matches the right side of a production rule*; and
  - whose *reduction* to the nonterminal on the left side of the production represents *one step along the reverse of a rightmost derivation;*
- Therefore, *not every substring (or more specifically, the leftmost substring)* that matches the right side of a production rule is *handle*.

E.g.:

G: $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id \mid (E)$

Input: $id_1 * id_2$

	$id_1 * id_2$	*matched substring*$\{id_1 , id_2\}$
$\Rightarrow$	$F * id_2$	*matched substring*$\{F , id_2\}$
$\Rightarrow$	$T * id_2$	*matched substring*$\{T , id_2\}$

In the next step, shall we reduce the leftmost substring $E \rightarrow T$ or $F \rightarrow id_2$?

Shift-Reduce Parsing

- A stack implementation of bottom-up parsing
  - **Shift** → Current input symbol is pushed onto the stack
  - **Reduce** → Right side of a production is replaced by the left side non-terminal in the stack.

- Shift zero or more input symbols onto the stack, until it is ready to reduce a string □ to a non-terminal A on top of the stack, if the grammar has production A → □.

- Repeat the process, until
  - It generate an error signal OR
  - Stack contains the start symbol and input is empty. Accept the input.

Shift-Reduce Parsing

G: $E \to E + T \mid T$
$T \to T * F \mid F$
$F \to id \mid (E)$

Input: $id_1 * id_2$

Stack	Input	Action
$\$$	$id_1 * id_2 \$$	Shift
$\$ \ id_1$	$* id_2 \$$	Reduce by $F \to id$
$\$ \ F$	$* id_2 \$$	Reduce by $T \to F$
$\$ \ T$	$* id_2 \$$	Shift
$\$ \ T *$	$id_2 \$$	Shift
$\$ \ T * id_2$	$\$$	Reduce by $F \to id$
$\$ \ T * F$	$\$$	Reduce by $T \to T*F$
$\$ \ T$	$\$$	Reduce by $E \to T$
$\$ \ E$	$\$$	Accept

Observe, handle is always at the top of stack.

Shift-Reduce Parsing: Few key points

- Four primary operations
  - **Shift** → Current input symbol is pushed onto the stack
  - **Reduce** → Right side of a production is replaced by the left side non-terminal on the stack.
  - **Accept** → Announce successful completion of parsing
  - **Error** → Discover a syntax error and call error handling mechanism.

- Handle always appear on top of the stack
- For an unambiguous grammar, for every right-sentential form there is exactly one handle.
  - Remember given $S \Rightarrow^* \alpha$,
    - If α contains non-terminals, it is called as a sentential form of G

Conflicts

- There are grammars for which shift-reduce parsing cannot be used.
- Shift-Reduce parser for such grammars may have a configuration where the parser cannot decide whether to
  - Shift the symbols onto the stack or Reduce the handle to a non-terminal OR
  - Reduce the handle with some non-terminal A or B.

- These situations are called **conflicts**.
  - *Shift/Reduce* conflict
  - *Reduce/Reduce* conflict

Conflicts: Example 1

- Ambiguous grammar can not have shift-reduce parser
- `stmt` → if `expr` then `stmt` |
  if `expr` then `stmt` else `stmt` |
  other

- Let the configuration of parser is

Shift

Stack **Input**

$.... if `expr` then `stmt` else .... $

Reduce

`stmt`

Shift/Reduce
conflict

Conflicts: Example 2

- stmt → id (param_list) | expr = expr
  param_list → param_list, param | param
  param → id
  expr → id (expr_list) | id
  expr_list → expr_list, expr | expr

- Let the configuration is

  Stack
  $.... id (id

  Reduce/Reduce conflict

  Input
  , id) .... $

  Reduce with param → id OR
  Reduce with expr → id

LR Parsers

LR Parser

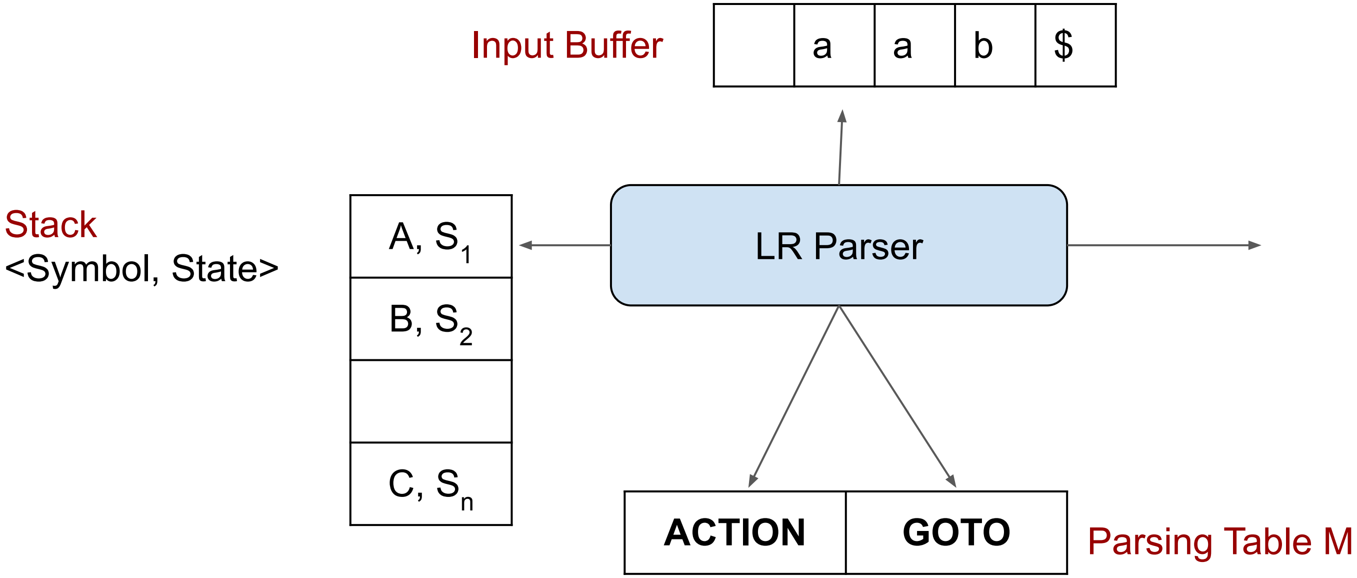
- LR(k) parsers are the most powerful and efficient shift-reduce parser
    - Left-to-right scanning, Right-most derivation (with k lookahead symbols)
    - In general, k = 1
    - In both LL(k) and LR(k), if k is omitted, it is assumed LL(1) and LR(1)
- A grammar for which we can construct a LR parser are called LR grammar
- Three main types of parse
    - Simple LR or SLR or LR(0)
    - Canonical LR or LR(1)
    - Look-ahead LR or LALR
- Parsing of all three parsers are similar, only their parsing tables are different

Why LR parsers?

- LR parsers can be constructed to recognize virtually all programming language constructs for which CFGs can be written.
- LR parsers are most general non-backtracking shift-reduce parser and yet its implementation is as efficient as others.
- An LR parser can detect a syntactic error as soon as it is possible to do on a left-to-right scan of the input
- Class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers or LL methods

$$\text{LL(1) grammars} \subset \text{LR(1) grammars}$$

LR parsing


Input Buffer
a
a
b
$
Stack
<Symbol, State>
A, S_1
B, S_2
C, S_n
LR Parser
Output
ACTION
GOTO
Parsing Table M

Configuration of LR parsing

- Each symbol on stack has an associated state.
- Initial stack configuration \$ S_0 (no symbol is associated with S_0)

$$(\$\ S_0\, X_1 S_1 \ldots X_m S_m, \qquad a_i\, a_{i+1} \ldots a_n\ \$\ )$$

Stack **Input**

- S_m and a_i decides the next parser action by consulting the parsing table M.

Configuration of LR parsing

- S_m and a_i decides the next parser action by consulting the parsing table M.
  - **Shift:**
    - Push a_i and its associated state S_i onto the stack

$(\$S_0X_1S_1 \ldots X_mS_m, \quad a_i\, a_{i+1} \ldots a_n\$) \quad \rightarrow \quad (\$S_0X_1S_1 \ldots X_mS_m a_i\, S_i, \quad a_{i+1} \ldots a_n\$)$

  - **Reduce:**
    - If $A \rightarrow X_{m-r-1}\, S_{m-r-1} \ldots\ldots X_mS_m$ is a handle
      - Pop $r = |X_{m-r-1}\, S_{m-r-1} \ldots\ldots X_mS_m|$ items from the stack
      - Push A and S onto the stack, where S = GOTO[S_{m-r}, A]

$(\$S_0X_1S_1 \ldots X_mS_m, \quad a_i\, a_{i+1} \ldots a_n\$) \quad \rightarrow \quad (\$S_0X_1S_1 \ldots X_{m-r}S_{m-r}\, A\, S, \quad a_i$

Construction of LR(k) parser

- Building a parser
  1. Build LR(k) automation
     a. Canonical set of “items”
  2. Build parsing table using LR(k) automation

- Once the parsing table is built, we can parse any given input string using LR(k) parsing algorithm.

Building LR(0) parser: Canonical set of items

Canonical set of “items” for LR(0) automation

- LR parser makes shift-reduce decision based on the states in an automation.
- Each state contains a set of items that reflects the progress in parsing.
- Collection of sets of LR(0) items are called canonical LR(0) collection.
- An LR(0) item (or simply item) of a grammar G is a production with a dot (.) at some position of the right side of the rule.
    - For the production A → XYZ, we have four items
        - A → .XYZ
        - A → X.YZ
        - A → XY.Z
        - A → XYZ.

The position of . indicates the amount of processing completed.

1. Parser has PROCESSED X on a portion of the input; and
2. HOPE to derive the rest of the input from YZ

(Dot) Closure of items

- To build the LR(0) automation, we need to find the closure of each item set(I)
- Let the grammar G:

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid id
\end{aligned}
$$

- Then, the dot closure of item $E \rightarrow .\, E + T$ is

$$
\boxed{
\begin{aligned}
E &\rightarrow .\, E + T \\
E &\rightarrow .\, T \\
T &\rightarrow .\, T * F \\
T &\rightarrow .\, F \\
F &\rightarrow .\, (E) \\
F &\rightarrow .\, id
\end{aligned}
}
\longleftarrow \text{Closure}(E \rightarrow .\, E + T)
$$

(Dot) Closure of items

Closure (I)

1. Add every item in I to Closure (I)
2. If $A \to \alpha.B\beta$ is in Closure(I) and $B \to \gamma$ is a production
   a. Add item $B \to .\gamma$ to Closure (I)
3. Repeat step 2, until no new items can be added to Closure (I).

Transition function GOTO()

- If Closure(I) has an item $A \to \alpha.B\beta$
  - GOTO (I, B) = Closure($A \to \alpha B.\beta$)

- Let Closure(I) = $\{[E \to .T], [E \to E. + T]\}$
  - GOTO (I, +) = $\{$ $[E \to E + . T]$,
    $[T \to . T * F]$
    $[T \to . F]$
    $[F \to . (E)]$
    $[F \to . id]$ $\}$

LR(0) Automation

- The state of the automation is defined by the $\mathrm{Closure}(I)$ of items
- The $\mathrm{GOTO}(I, X)$ function defines the transition from state I on symbol X

- For every grammar, augment a production $S' \to S$, if S was the starting symbol.
  - S' becomes new start symbol
  - $S' \to S.$ signifies the acceptance of the input.

Computation of the canonical LR(0) collection

Items(G')

1. C = Closure({[$S' \rightarrow .S$]})
2. Repeat
   a. For each set of items I in C
      i. For each grammar symbol X
         1. If GOTO(I, X) is not empty and not in C
            a. Add GOTO(I, X) to C
3. Until no new sets of items are added to C

LR(0) Automation

0: $E' \rightarrow E$	**1:** $E \rightarrow E + T$	**2:** $E \rightarrow T$	**3:** $T \rightarrow T * F$
4: $T \rightarrow F$	**5:** $F \rightarrow (E)$	**6:** $F \rightarrow id$	

$E' \rightarrow .E$

$E \rightarrow .E + T$

$E \rightarrow .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

LR(0) Automation

0: $E' \rightarrow E$	**1:** $E \rightarrow E + T$	**2:** $E \rightarrow T$	**3:** $T \rightarrow T * F$
4: $T \rightarrow F$	**5:** $F \rightarrow (E)$	**6:** $F \rightarrow id$	

$E' \rightarrow .E$
$E \rightarrow .E + T$
$E \rightarrow .T$
$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$
I_0

$\xrightarrow{E}$

$E' \rightarrow E\,.$
$E \rightarrow E\,. + T$

LR(0) Automation

0: $E' \to E$	**1:** $E \to E + T$	**2:** $E \to T$	**3:** $T \to T * F$
4: $T \to F$	**5:** $F \to (E)$	**6:** $F \to id$	

I_0:

$E' \to .E$

$E \to .E + T$

$E \to .T$

$T \to .T * F$

$T \to .F$

$F \to .(E)$

$F \to .id$

$I_0 \xrightarrow{E} I_1$

I_1:

$E' \to E\,.$

$E \to E\,. + T$

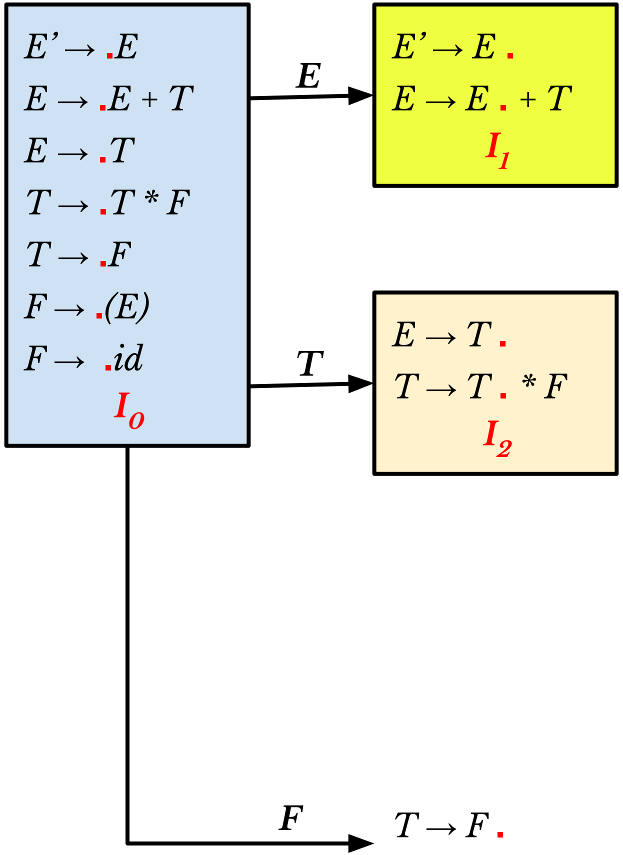
$I_0 \xrightarrow{T}$

$E \to T\,.$

$T \to T\,. * F$

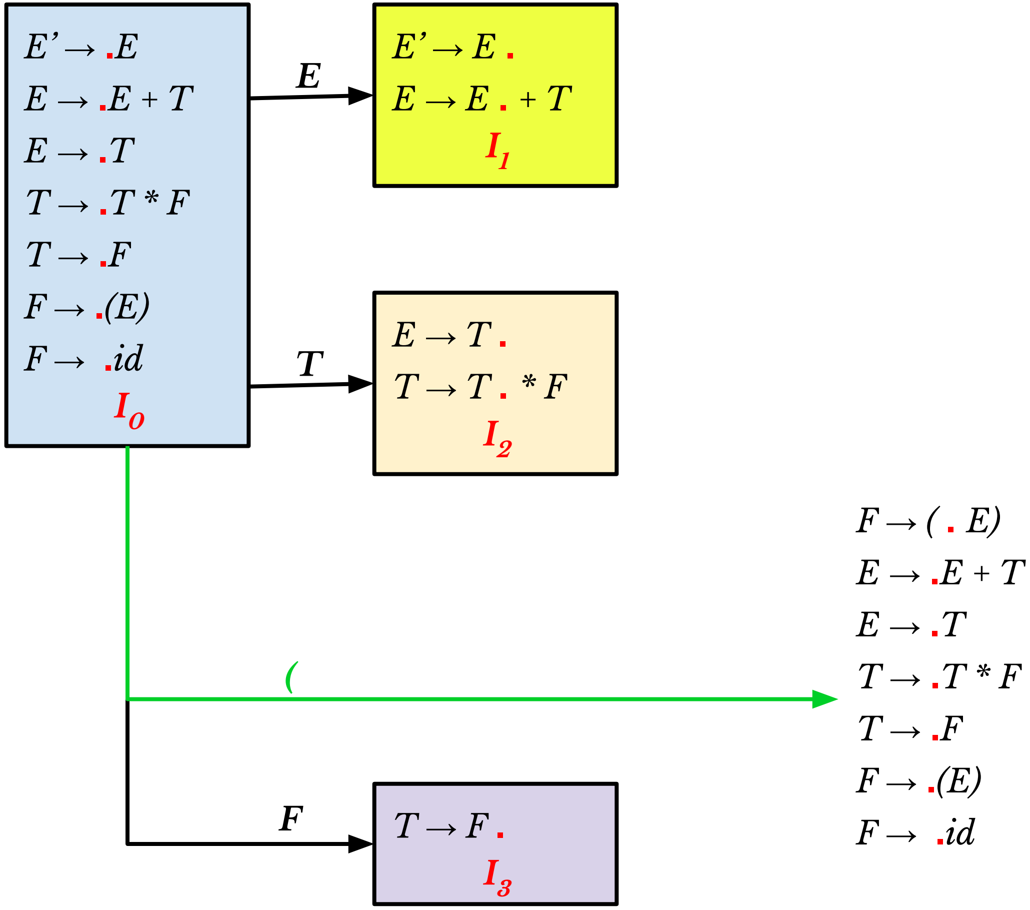
LR(0) Automation

0: $E' \rightarrow E$ **1:** $E \rightarrow E + T$ **2:** $E \rightarrow T$ **3:** $T \rightarrow T * F$
4: $T \rightarrow F$ **5:** $F \rightarrow (E)$ **6:** $F \rightarrow id$



LR(0) Automation

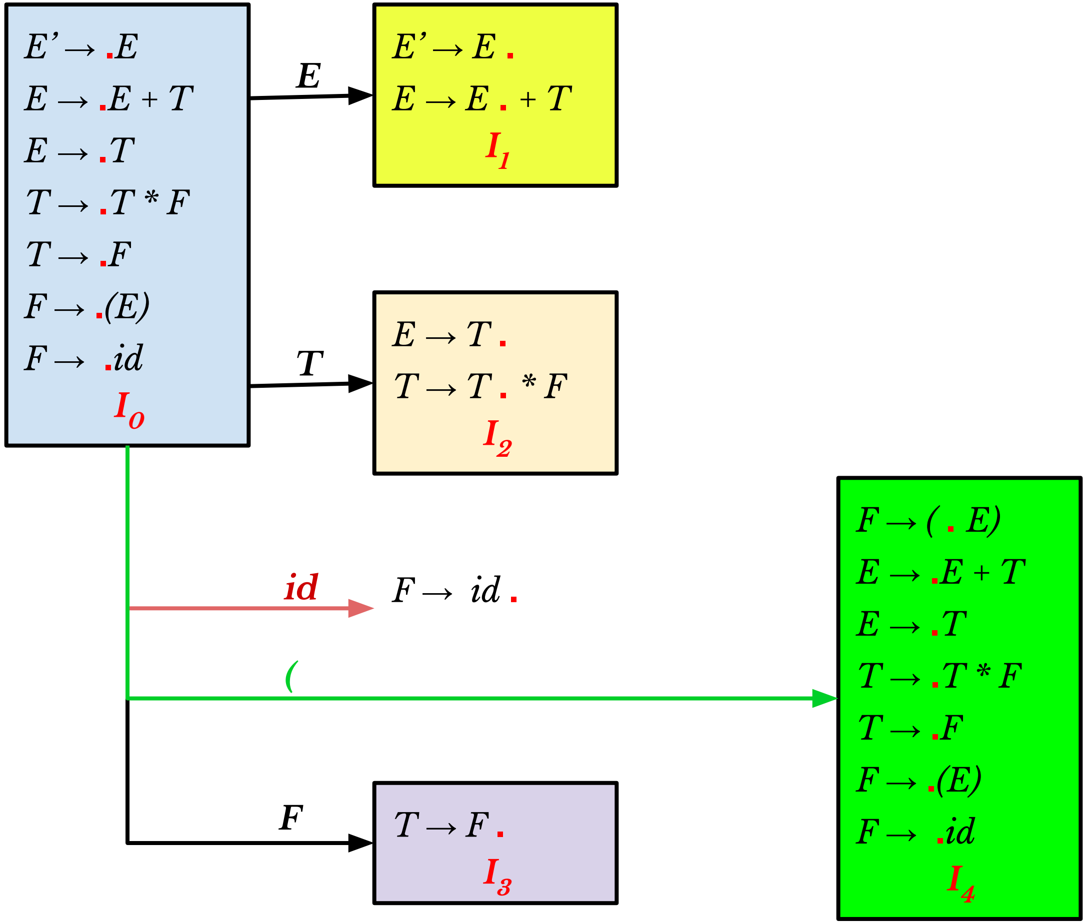
0: $E' \rightarrow E$	1: $E \rightarrow E + T$	2: $E \rightarrow T$	3: $T \rightarrow T * F$
4: $T \rightarrow F$	5: $F \rightarrow (E)$	6: $F \rightarrow id$	



LR(0) Automation

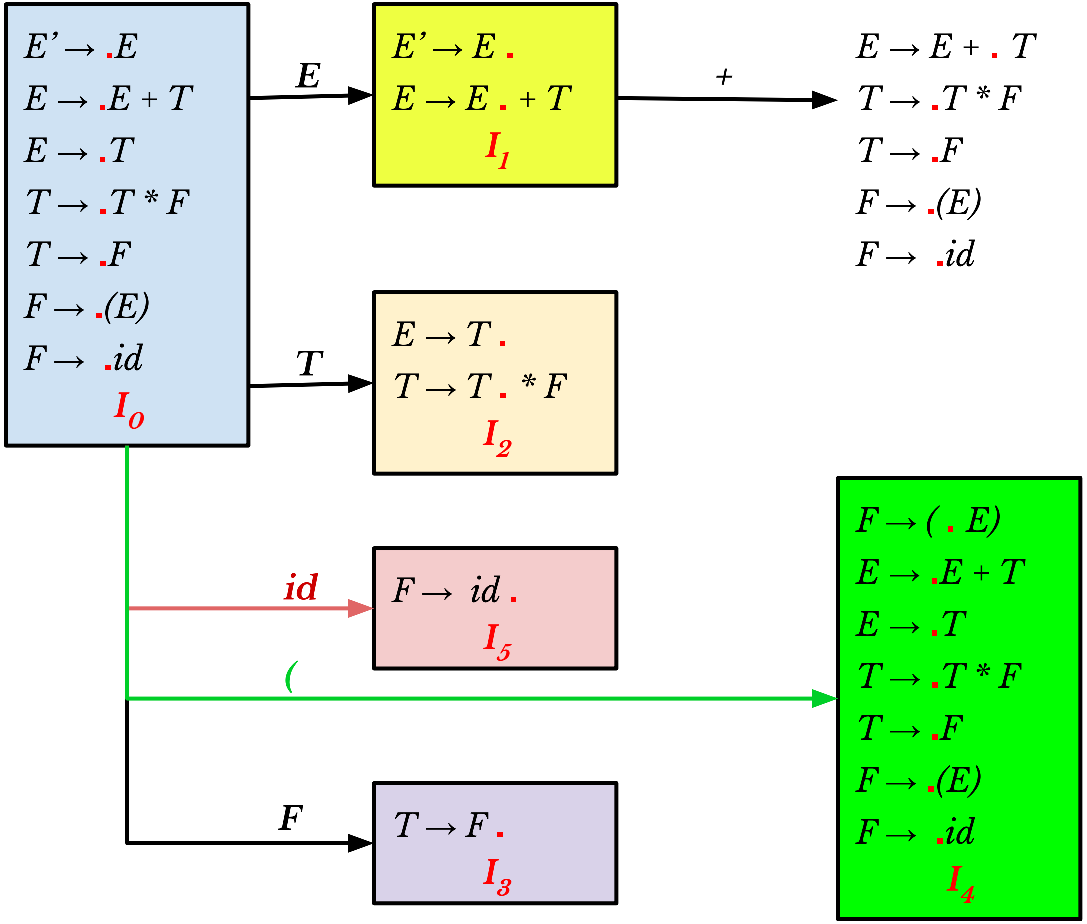
0: $E' \to E$ 1: $E \to E + T$ 2: $E \to T$ 3: $T \to T * F$

4: $T \to F$ 5: $F \to (E)$ 6: $F \to id$



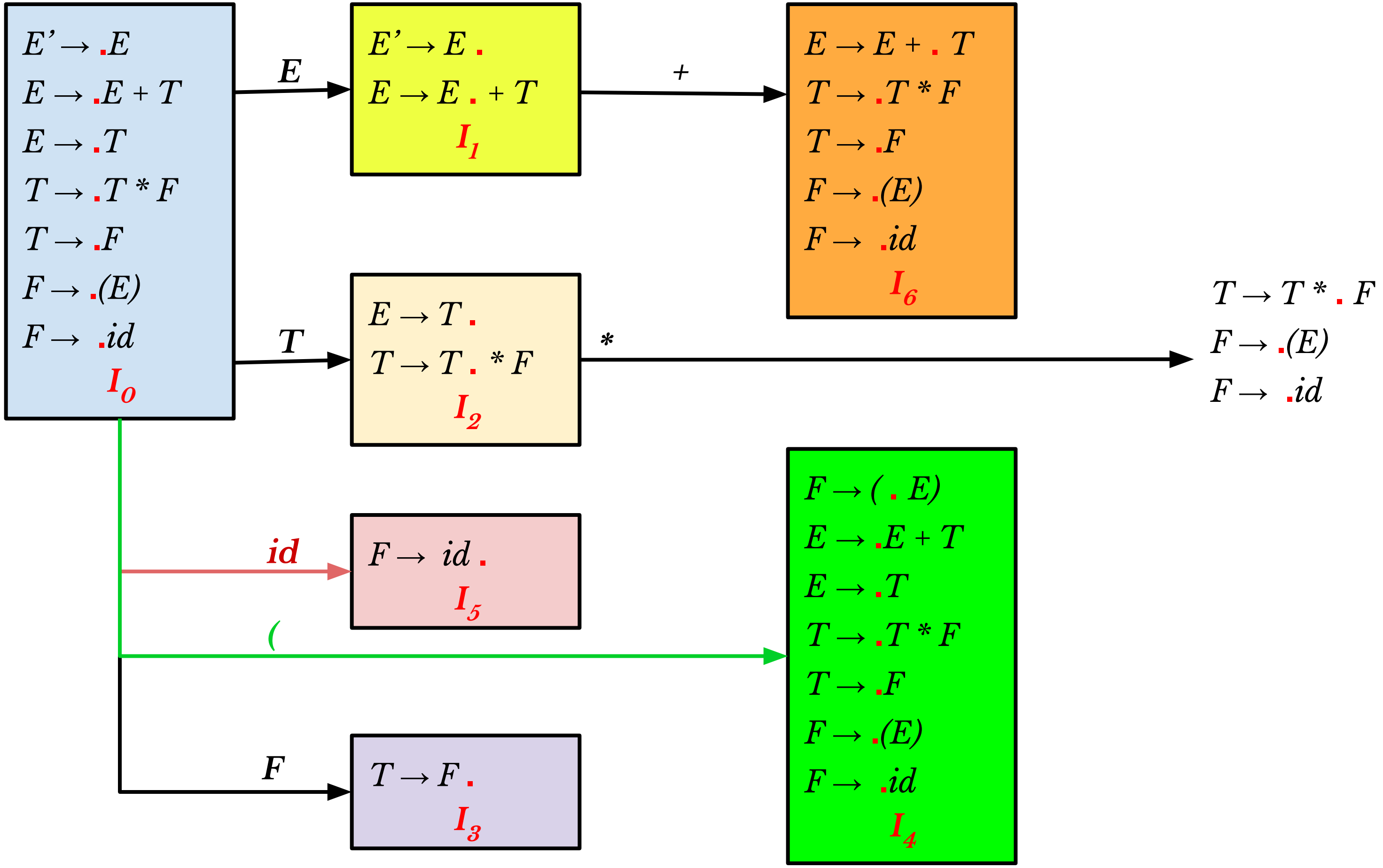
LR(0) Automation

0: $E' \rightarrow E$	**1:** $E \rightarrow E + T$	**2:** $E \rightarrow T$	**3:** $T \rightarrow T * F$
4: $T \rightarrow F$	**5:** $F \rightarrow (E)$	**6:** $F \rightarrow id$	



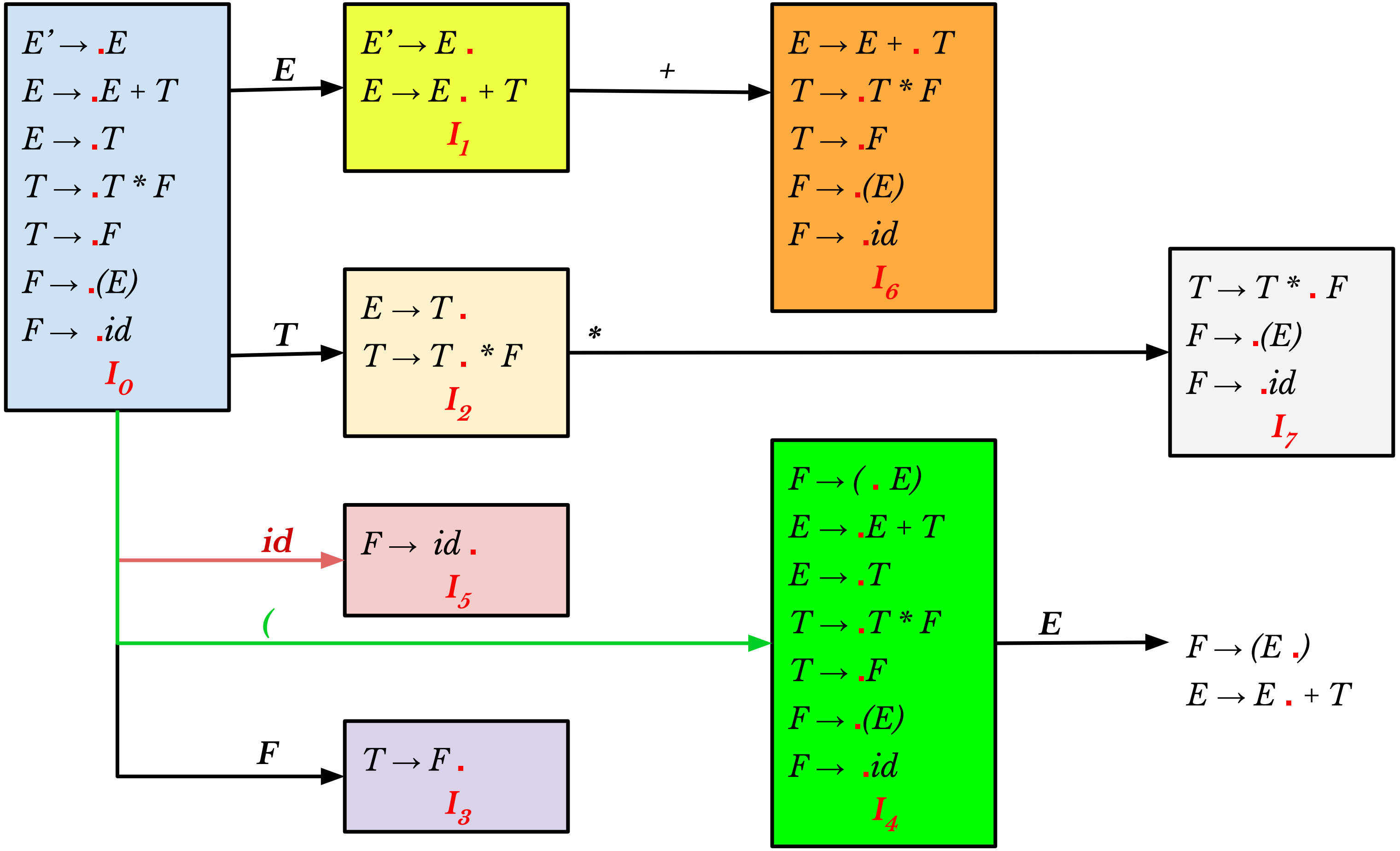
LR(0) Automation

0: $E' \rightarrow E$	**1:** $E \rightarrow E + T$	**2:** $E \rightarrow T$	**3:** $T \rightarrow T * F$
4: $T \rightarrow F$	**5:** $F \rightarrow (E)$	**6:** $F \rightarrow id$	



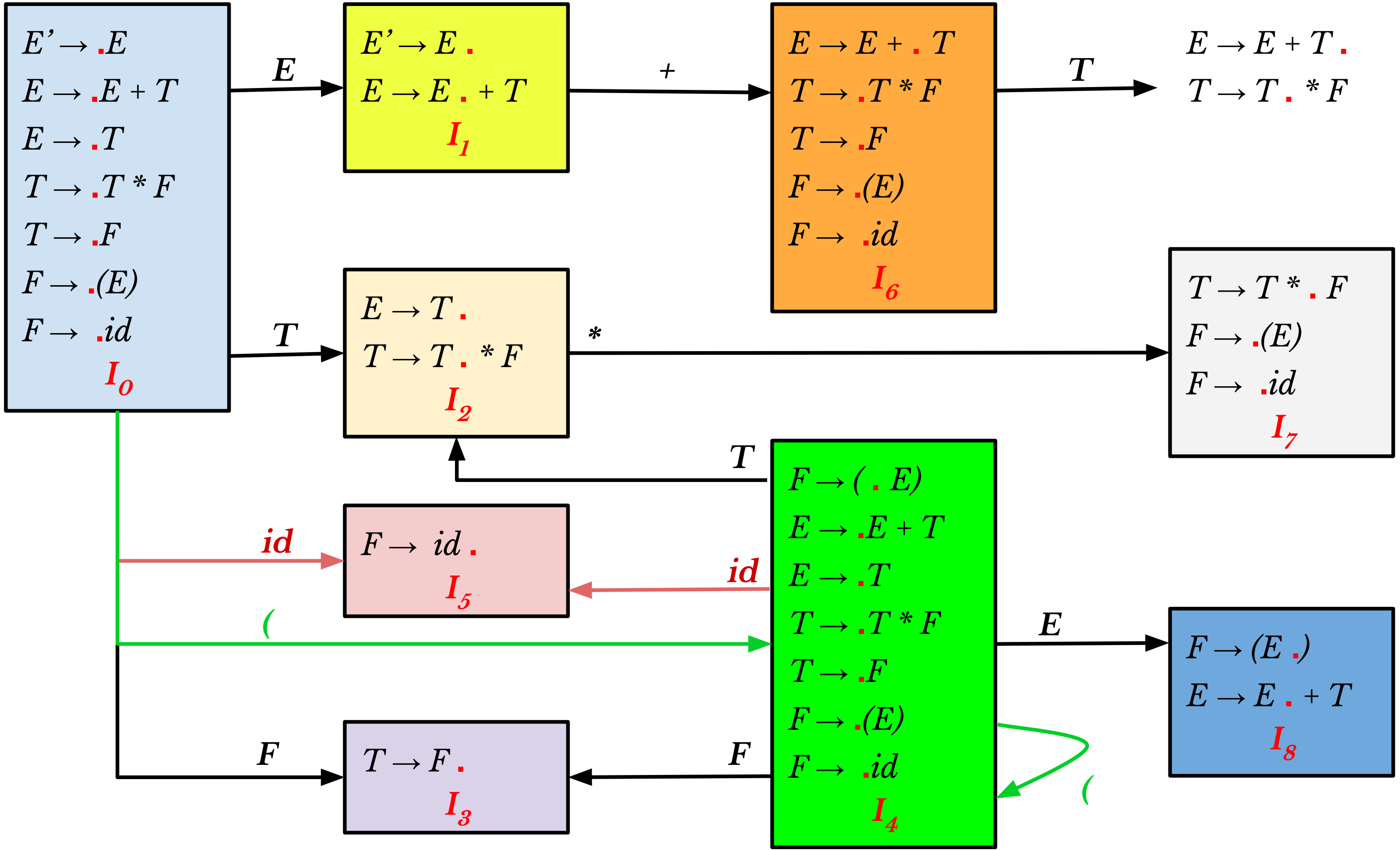
LR(0) Automation

0: $E' \to E$	**1:** $E \to E + T$	**2:** $E \to T$	**3:** $T \to T * F$
4: $T \to F$	**5:** $F \to (E)$	**6:** $F \to id$	



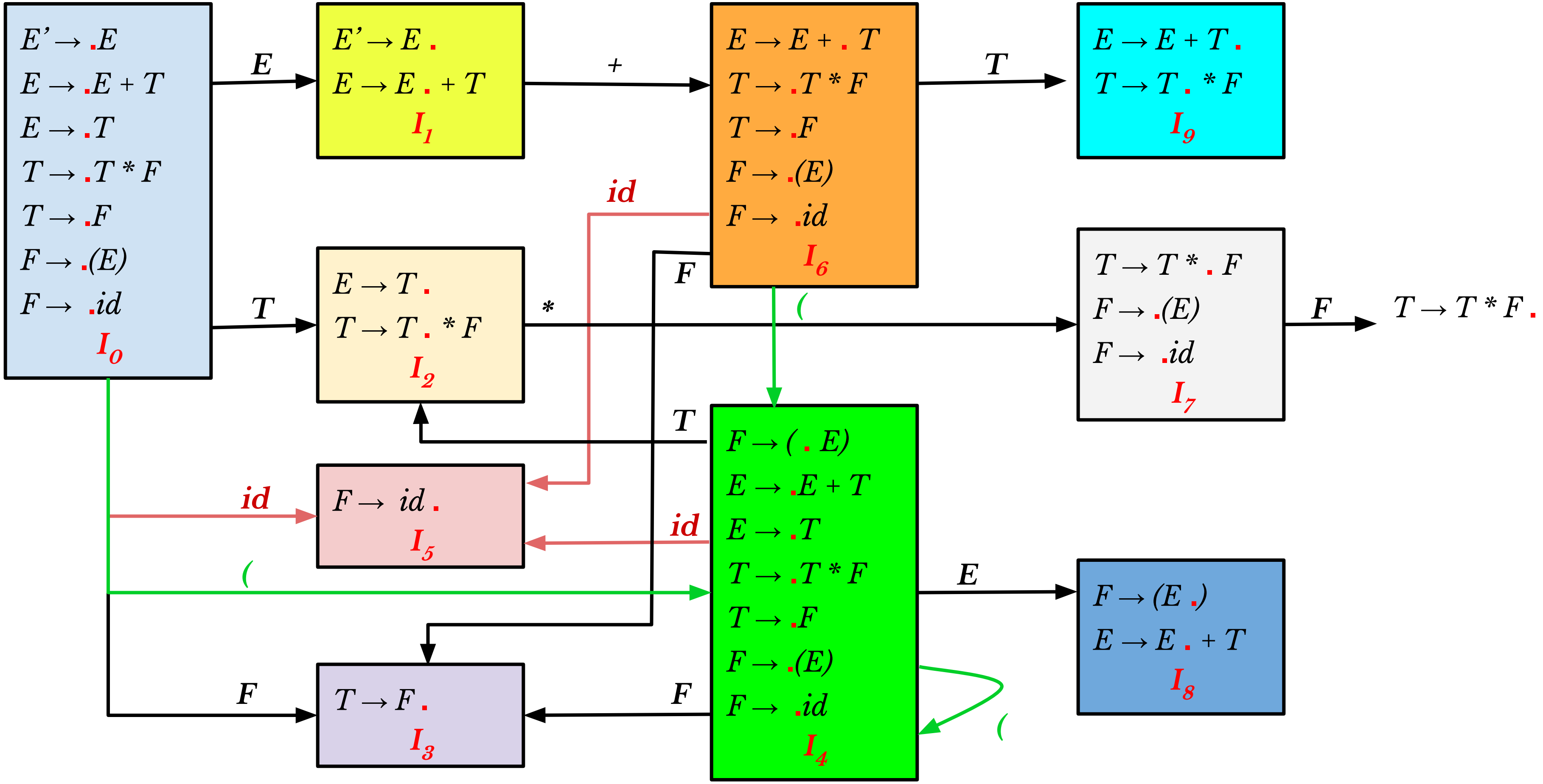
LR(0) Automation

0: $E' \to E$ 1: $E \to E + T$ 2: $E \to T$ 3: $T \to T * F$
4: $T \to F$ 5: $F \to (E)$ 6: $F \to id$



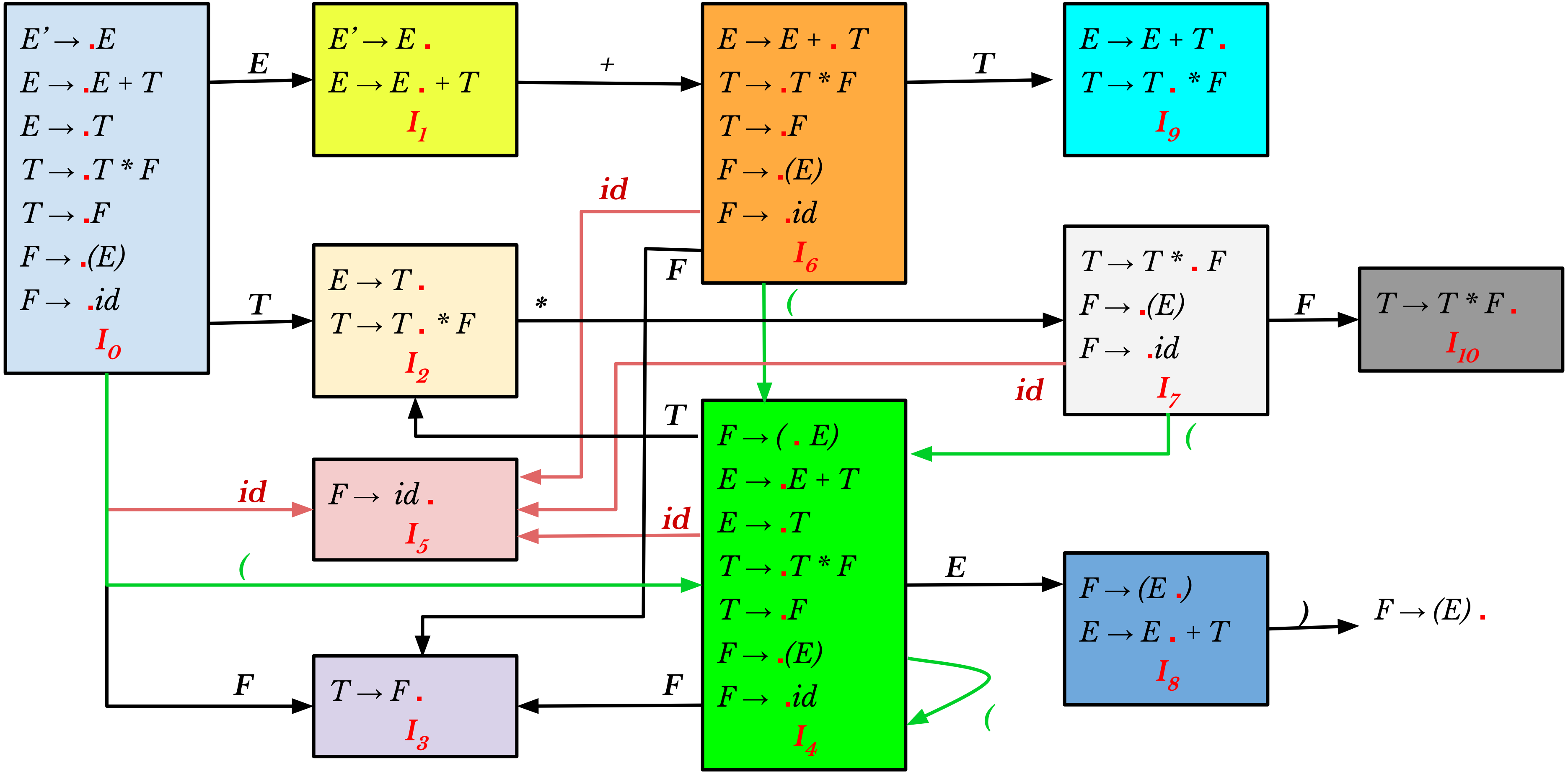
LR(0) Automation

0: $E' \to E$	1: $E \to E + T$	2: $E \to T$	3: $T \to T * F$
4: $T \to F$	5: $F \to (E)$	6: $F \to id$	



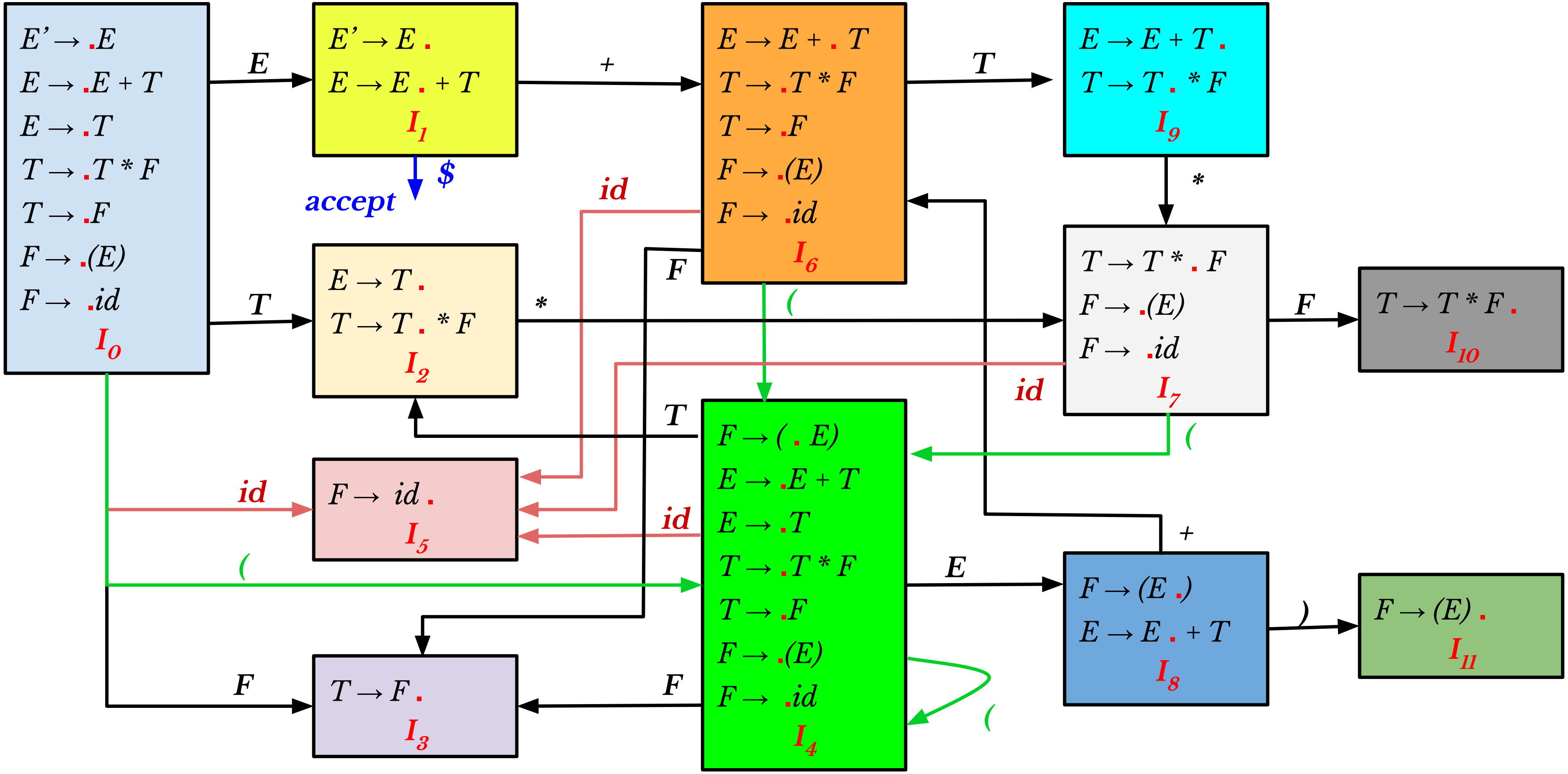
LR(0) Automation

0: $E' \to E$	**1:** $E \to E + T$	**2:** $E \to T$	**3:** $T \to T * F$
4: $T \to F$	**5:** $F \to (E)$	**6:** $F \to id$	



LR(0) Automation

0: $E' \to E$	**1:** $E \to E + T$	**2:** $E \to T$	**3:** $T \to T * F$
4: $T \to F$	**5:** $F \to (E)$	**6:** $F \to id$	



Building LR(0) parser: Parsing table

Constructing SLR parsing table

- Remember, LR parsing table has two parts
  - Action: Takes only terminals
  - GOTO: Takes only non-terminals

SLR-Table(G')

1. Construct LR(0) collection for the grammar G'
2. Let I_i represents state S_i, then the parsing action for state i are as follows
   a. If $[A \to \alpha.a\beta]$ is in I_i and GOTO(I_i, a) = I_j
      i. Action[i , a] = "shift j"
   b. If $[A \to \alpha.]$ is in I_i
      i. Action[i , a] = "reduce $A \to \alpha$" for all $a \in$ FOLLOW(A)
   c. If $[S' \to S.]$ is in I_i
      i. Action[i , \$] = "accept"
3. For all non-terminals A,
   a. if GOTO(I_i, A) = I_j,
      i. GOTO[i, A] = j

SLR parsing table

GOTO(S, X): Transition from state S to a new state on non-terminal symbol X

For all transitions on non-terminals in state 0

GOTO(0, E) = 1
GOTO(0, T) = 2
GOTO(0, F) = 3

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0							1	2	3
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									

SLR parsing table

GOTO(S, X): Transition from state S to a new state on non-terminal symbol X

For all transitions on non-terminals in state 4

GOTO(4, E) = 8
GOTO(4, T) = 2
GOTO(4, F) = 3

For all transitions on non-terminals in state 6

GOTO(6, T) = 9
GOTO(6, F) = 3

For all transitions on non-terminals in state 7

GOTO(7, F) = 10

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0							1	2	3
1									
2									
3									
4							8	2	3
5									
6								9	3
7									10
8									
9									
10									
11									

SLR parsing table

If $[A \rightarrow \alpha.a\beta]$ is in I_i and GOTO(I_i, a) = I_j then

Action[i , a] = "shift j"

Note: a is terminal

For all transitions on terminals in state 0

Action[0, id] = "shift 5" or "s5"

Action[0, (] = "s4"

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1									
2									
3									
4							8	2	3
5									
6								9	3
7									10
8									
9									
10									
11									

SLR parsing table

If $[A \rightarrow \alpha.a\beta]$ is in I_i and GOTO(I_i, a) = I_j then

Action[i , a] = "shift j"

Note: a is terminal

For all transitions on terminals in state 1

Action[1, +] = "s6"

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6							
2									
3									
4							8	2	3
5									
6								9	3
7									10
8									
9									
10									
11									

SLR parsing table

If $[A \rightarrow \alpha.a\beta]$ is in I_i and GOTO(I_i, a) = I_j then

Action[i , a] = "shift j"

Note: a is terminal

For all transitions on terminals in state 2

Action[2, *] = "s7"

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6							
2			**s7**						
3									
4							8	2	3
5									
6								9	3
7									10
8									
9									
10									
11									

SLR parsing table

If $[A \rightarrow \alpha.a\beta]$ is in I_i and GOTO(I_i, a) = I_j then

Action[i , a] = "shift j"

Note: a is terminal

For all transitions on terminals in state 3

None

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6							
2			s7						
3									
4							8	2	3
5									
6								9	3
7									10
8									
9									
10									
11									

SLR parsing table

If $[A \rightarrow \alpha.a\beta]$ is in I_i and GOTO(I_i, a) = I_j then

Action[i , a] = "shift j"

Note: a is terminal

For all transitions on terminals in state 4

Action[4, id] = "s5"

Action[4, ( ] = "s4"

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6							
2			s7						
3									
4	**s5**			**s4**			8	2	3
5									
6								9	3
7									10
8									
9									
10									
11									

SLR parsing table

If $[A \rightarrow \alpha.a\beta]$ is in I_i and GOTO(I_i, a) = I_j then

Action[i , a] = "shift j"

Note: a is terminal

For all transitions on terminals in state 5

None

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6							
2			s7						
3									
4	s5			s4			8	2	3
5									
6								9	3
7									10
8									
9									
10									
11									

SLR parsing table

If $[A \rightarrow \alpha.a\beta]$ is in I_i and GOTO(I_i, a) = I_j
then

Action[i , a] = "shift j"

Note: a is terminal

For all transitions on terminals in state 6

Action[6, id] = "s5"

Action[6, ( ] = "s4"

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6							
2			s7						
3									
4	s5			s4			8	2	3
5									
6	**s5**			**s4**				9	3
7									10
8									
9									
10									
11									

SLR parsing table

If $[A \rightarrow \alpha.a\beta]$ is in I_i and GOTO(I_i, a) = I_j then

Action[i , a] = “shift j”

Note: a is terminal

For all transitions on terminals in state 7

Action[7, id] = “s5”

Action[7, ( ] = “s4”

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6							
2			s7						
3									
4	s5			s4			8	2	3
5									
6	s5			s4				9	3
7	**s5**			**s4**					10
8									
9									
10									
11									

SLR parsing table

If $[A \rightarrow \alpha.a\beta]$ is in I_i and GOTO(I_i, a) = I_j then

Action[i , a] = "shift j"

Note: a is terminal

For all transitions on terminals in state 8

Action[8, +] = "s6"

Action[8,)] = "s11"

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6							
2			s7						
3									
4	s5			s4			8	2	3
5									
6	s5			s4				9	3
7	s5			s4					10
8		**s6**			**s11**				
9									
10									
11									

SLR parsing table

If $[A \rightarrow \alpha.a\beta]$ is in I_i and GOTO(I_i, a) = I_j then

Action[i , a] = "shift j"

Note: a is terminal

For all transitions on terminals in state 9

Action[9, *] = "s7"

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6							
2			s7						
3									
4	s5			s4			8	2	3
5									
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9			s7						
10									
11									

SLR parsing table

If $[A \rightarrow \alpha.a\beta]$ is in I_i and GOTO(I_i, a) = I_j then

Action[i , a] = “shift j”

Note: a is terminal

For all transitions on terminals in state 10

None

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6							
2			s7						
3									
4	s5			s4			8	2	3
5									
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9			s7						
10									
11									

SLR parsing table

If $[A \rightarrow \alpha.a\beta]$ is in I_i and GOTO(I_i, a) = I_j then

Action[i , a] = "shift j"

Note: a is terminal

For all transitions on terminals in state 11

None

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6							
2			s7						
3									
4	s5			s4			8	2	3
5									
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9			s7						
10									
11									

SLR parsing table

If $[A \to \alpha.]$ is in I_i
Action$[i, a]$ = "reduce $A \to \alpha$" for all $a \in$ FOLLOW(A)

Statse 0, 4, 6, 7 and 8 does not have any such production

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6							
2			s7						
3									
4	s5			s4			8	2	3
5									
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9			s7						
10									
11									

SLR parsing table

If $[A \to \alpha.]$ is in I_i
Action$[i, a]$ = "reduce $A \to \alpha$" for all a $\in$ FOLLOW(A)

For all terminals in Follow(E→T) in state 2
Action[2, +] = "reduce E→T" or "r2"
Action[2,)] = "r2"
Action[2, $] = "r2"

r2 means reduction by production no 2.
[Remember, we numbered the productions]

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6							
2		**r2**	s7		**r2**	**r2**			
3									
4	s5			s4			8	2	3
5									
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9			s7						
10									
11									

SLR parsing table

If $[A \to \alpha.]$ is in I_i

Action$[i, a]$ = "reduce $A \to \alpha$" for all $a \in$ FOLLOW(A)

For all terminals in Follow(T→F) in state 3

Action[3, +] = "r4"

Action[3, *] = "r4"

Action[3,)] = "r4"

Action[3, $] = "r4"

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6							
2		r2	s7		r2	r2			
3		**r4**	**r4**		**r4**	**r4**			
4	s5			s4			8	2	3
5									
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9			s7						
10									
11									

SLR parsing table

If $[A \rightarrow \alpha.]$ is in I_i

Action$[i, a]$ = "reduce $A \rightarrow \alpha$" for all $a \in$ FOLLOW(A)

For all terminals in Follow(F→id) in state 5

Action[5, +] = "r6"

Action[5, *] = "r6"

Action[5,)] = "r6"

Action[5, $] = "r6"

State	**Action**						**GOTO**		
	id	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
0	s5			s4			1	2	3
1		s6							
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			s7						
10									
11									

SLR parsing table

If $[A \rightarrow \alpha.]$ is in I_i

Action$[i\ , a]$ = "reduce $A \rightarrow \alpha$" for all a $\in$ FOLLOW(A)

For all terminals in Follow(E→E+T) in state 9

Action[9, +] = "r1"

Action[9,)] = "r1"

Action[9, $] = "r1"

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6							
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									
11									

SLR parsing table

If $[A \rightarrow \alpha\textbf{.}]$ is in I_i
Action$[i, a]$ = "reduce $A \rightarrow \alpha$" for all a $\in$ FOLLOW(A)

For all terminals in Follow(T→T*F) in state 10
Action[10, +] = "r3"
Action[10, *] = "r3"
Action[10,)] = "r3"
Action[10, $] = "r3"

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6							
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									

SLR parsing table

If $[A \rightarrow \alpha.]$ is in I_i

Action$[i\,, a]$ = "reduce $A \rightarrow \alpha$" for all $a \in$ FOLLOW(A)

For all terminals in Follow(F→(E)) in state 11

Action[11, +] = "r5"

Action[11, *] = "r5"

Action[11,)] = "r5"

Action[11, $] = "r5"

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6							
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**			

SLR parsing table

If $[S' \rightarrow S.]$ is in I_i

Action$[i$, $]$ = "accept"

Action[1, $] = "accept"

All empty entries are "error" case.

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			

LR(0) parser: Parsing an input string

SLR parsing algorithm

1. Let a be the first symbol in $w\$$
2. Repeat
   a. Let s be the state on top of the stack
   b. If Action[s, a] == $s\#t$
      i. Push t on to the stack
      ii. Let a be the next symbol
   c. Else if Action[s, a] == reduce $A \to B$
      i. Pop $|B|$ symbols off the stack
      ii. Push GOTO[t, A] on to the stack
      iii. Output production $A \to B$
   d. Else if Action[s, a] == "accept"
      i. Halt
   e. Else
      i. Error : Call error handler

SLR parsing

Input: id * id + id

Stack	Symbol	Input	Action
0		id * id + id $	Shift
0 5	id	* id + id $	Reduce F → id
0 3	F	* id + id $	Reduce T → F
0 2	T	* id + id $	Shift
0 2 7	T *	id + id $	Shift
0 2 7 5	T * id	+ id $	Reduce F → id
0 2 7 10	T * F	+ id $	Reduce T → T * F
0 2	T	+ id $	Reduce E → T
0 1	E	+ id $	Shift
0 1 6	E +	id $	Shift
0 1 6 5	E + id	$	Reduce F → id
0 1 6 3	E + F	$	Reduce T → F
0 1 6 9	E + T	$	Reduce E → E + T
0 1	E	$	**Accept**

LR(0) Automation: Another example

Grammar G: $S \rightarrow L = R \mid R$

$L \rightarrow {}^{*}R \mid id$

$R \rightarrow L$

Construct SLR parser for the above grammar

I_0: $S' \rightarrow .\ S$
$S \rightarrow .\ L = R$
$S \rightarrow .\ R$
$L \rightarrow .\ {}^{*}\ R$
$L \rightarrow .\ id$
$R \rightarrow .\ L$

I_1: $S' \rightarrow S\ .$

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

I_3: $S \rightarrow R\ .$

I_4: $L \rightarrow {}^{*}\ .\ R$
$R \rightarrow .\ L$
$L \rightarrow .\ {}^{*}\ R$
$L \rightarrow .\ id$

I_5: $L \rightarrow id\ .$

I_6: $S \rightarrow L = .\ R$
$R \rightarrow .\ L$
$L \rightarrow .\ {}^{*}\ R$
$L \rightarrow .\ id$

I_7: $L \rightarrow {}^{*}\ R\ .$

I_8: $R \rightarrow L\ .$

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

Parsing table entry for state 2

Action[2, =] = "shift 6" or "reduce $R \rightarrow L$" ?

SLR(1) grammar

- If any cell in the parsing table has multiple entries, then
  - Grammar is not SLR(1) or LR(0)

Grammar G:

$$S \rightarrow L = R \mid R$$
$$L \rightarrow *R \mid id$$
$$R \rightarrow L$$

Not SLR(1)

- Every SLR(1) grammar is unambiguous.
- But, there are many unambiguous grammar that are not SLR(1)

LR(0) Automation: Another example - 2

Grammar G: $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

Construct SLR parser for the above grammar

I_0:
$S' \rightarrow . S$
$S \rightarrow . AaAb$
$S \rightarrow . BbBa$
$A \rightarrow .$
$B \rightarrow .$

Parsing table entry for state I_0

Action[0, a] = "reduce $A \rightarrow \varepsilon$" or "reduce $B \rightarrow \varepsilon$" ?

Follow(A) = {a, b}
Follow(B) = {a, b}

More Powerful LR Parsers

LR(1) and LALR parsers

- Canonical LR or LR(1) parser
  - Makes full use of lookahead symbols, i.e., both First() and Follow() symbols
    - Recall, LR(0) uses only Follow() symbols

- Lookahead LR or LALR parser
  - Inclusion of lookahead symbols in LR(0) sets of items
    - Can handle more grammars than SLR method
    - LALR tables are no bigger than the SLR tables
  - Fewer states than typical LR(1) parser

SLR vs LR(1)

- Reduction during parsing in SLR
  - $\text{Stack}(\beta\alpha) \Rightarrow \text{Stack}(\beta A)$ if we had $[A \rightarrow \alpha.]$
- What if, βA is not followed by a in right-sentential form
  - Reduction $A \rightarrow \alpha$ is invalid
- E.g.:
  - Grammar G: S → L = R | R
    L → *R | id
    R → L
  - If we apply "reduce *R → L*" on Action[2, =]
    "*L = …*" ⇒ "*R = …*"
  - However, there is no right-sentential form of the grammar the begins with "*R = …*"

Building LR(1) parser: Canonical set of items

Canonical LR(1) Items

- General form of an LR(1) item is $[A \rightarrow \alpha.\beta, a]$
  - $A \rightarrow \alpha\beta$ is a production, and
  - a is a lookahead terminal symbol or endmarker \$

- Lookahead symbol has no effect in an item of the form $[A \rightarrow \alpha.\beta, a]$, where β is not ε

- Lookahead symbol is required during the reduction only
  - We reduce $[A \rightarrow \alpha. , a]$, only if the next input symbol is a

Computation of the canonical LR(1) collection

Items(G')

1. C = Closure({[$S' \rightarrow .S$, \$]})
2. Repeat
   a. For each set of items I in C
      i. For each grammar symbol X
         1. If GOTO(I, X) is not empty and not in C
            i. Add GOTO(I, X) to C
3. Until no new sets of items are added to C

Closure(I)

1. Repeat
   a. For each items $[A \rightarrow \alpha.B\beta, a]$ in I
      i. For each production $B \rightarrow \gamma$ in G
         1. For each terminal b in FIRST(βa)
            i. Add $[B \rightarrow .\gamma, b]$
2. Until no new items are added to I

GOTO(I, X)

1. For each items $[A \rightarrow \alpha\, .\, X\beta, a]$ in I
   a. Add item $[A \rightarrow \alpha X\, .\, \beta, a]$ to J
2. return Closure(J)

LR(1) Automation

0: $S' \rightarrow S$ **1:** $S \rightarrow CC$ **2:** $C \rightarrow c\ C$ **3:** $C \rightarrow d$

$S' \rightarrow .S,\ \$$

$S \rightarrow .CC,\ \$$

$C \rightarrow .cC,\ \mathrm{c/d}$

$C \rightarrow .d,\ \mathrm{c/d}$

LR(1) Automation

0: $S' \rightarrow S$ **1:** $S \rightarrow CC$ **2:** $C \rightarrow c\ C$ **3:** $C \rightarrow d$

$S' \rightarrow .S,\ \$$
$S \rightarrow .CC,\ \$$
$C \rightarrow .cC,\ \text{c/d}$
$C \rightarrow .d,\ \text{c/d}$
I_0

$\xrightarrow{S}$ $S' \rightarrow S\ .,\ \$$

LR(1) Automation

0: $S' \to S$ **1:** $S \to CC$ **2:** $C \to c\, C$ **3:** $C \to d$

$S' \to .S,\ \$$
$S \to .CC,\ \$$
$C \to .cC,\ \mathrm{c/d}$
$C \to .d,\ \mathrm{c/d}$
I_0

$\xrightarrow{S}$

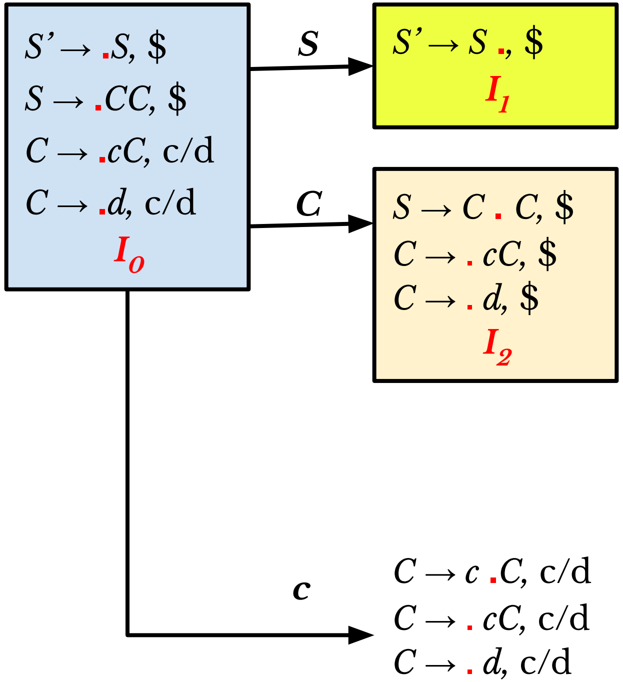
$S' \to S\, .,\ \$$
I_1

$\xrightarrow{C}$

$S \to C\ .\ C,\ \$$
$C \to .\ cC,\ \$$
$C \to .\ d,\ \$$

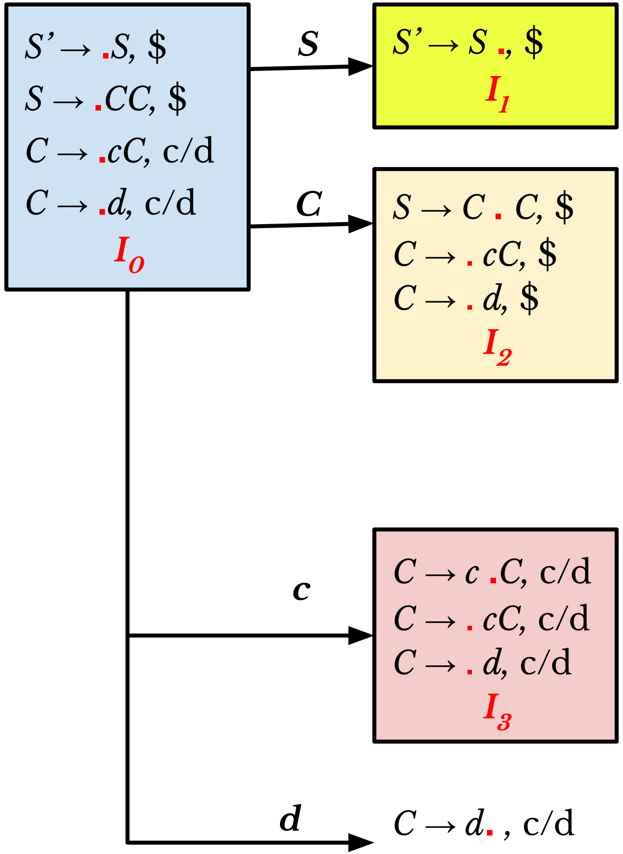
LR(1) Automation

0: $S' \rightarrow S$ **1:** $S \rightarrow CC$ **2:** $C \rightarrow c\,C$ **3:** $C \rightarrow d$



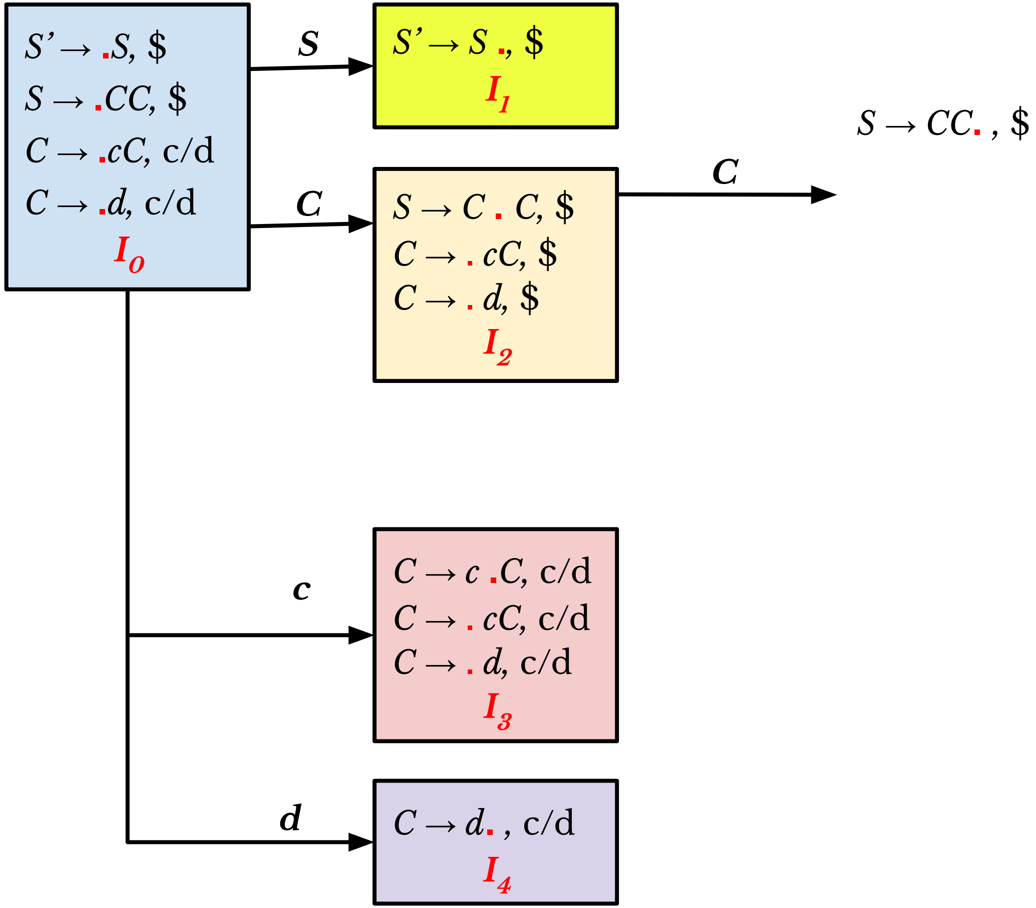
LR(1) Automation

0: $S' \rightarrow S$ **1:** $S \rightarrow CC$ **2:** $C \rightarrow c\ C$ **3:** $C \rightarrow d$



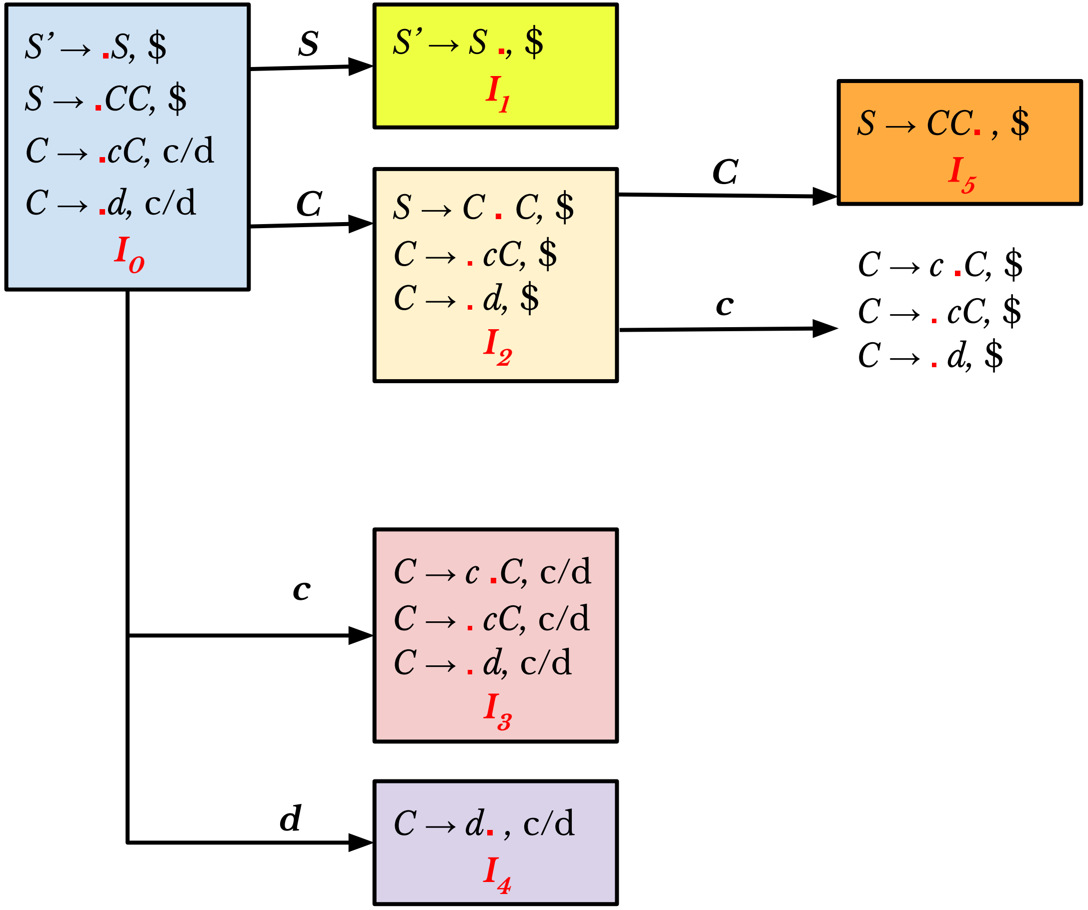
LR(1) Automation

0: $S' \rightarrow S$ **1:** $S \rightarrow CC$ **2:** $C \rightarrow c\ C$ **3:** $C \rightarrow d$



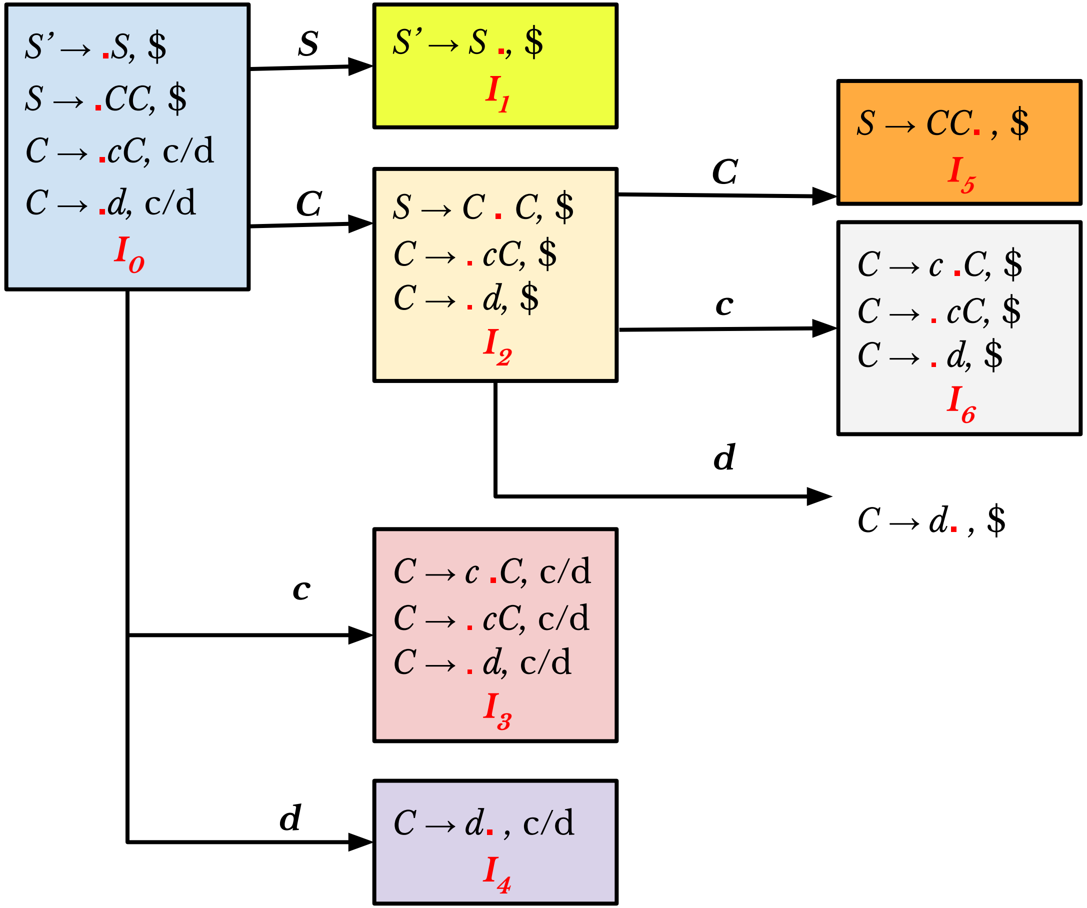
LR(1) Automation

0: $S' \to S$ **1:** $S \to CC$ **2:** $C \to c\ C$ **3:** $C \to d$



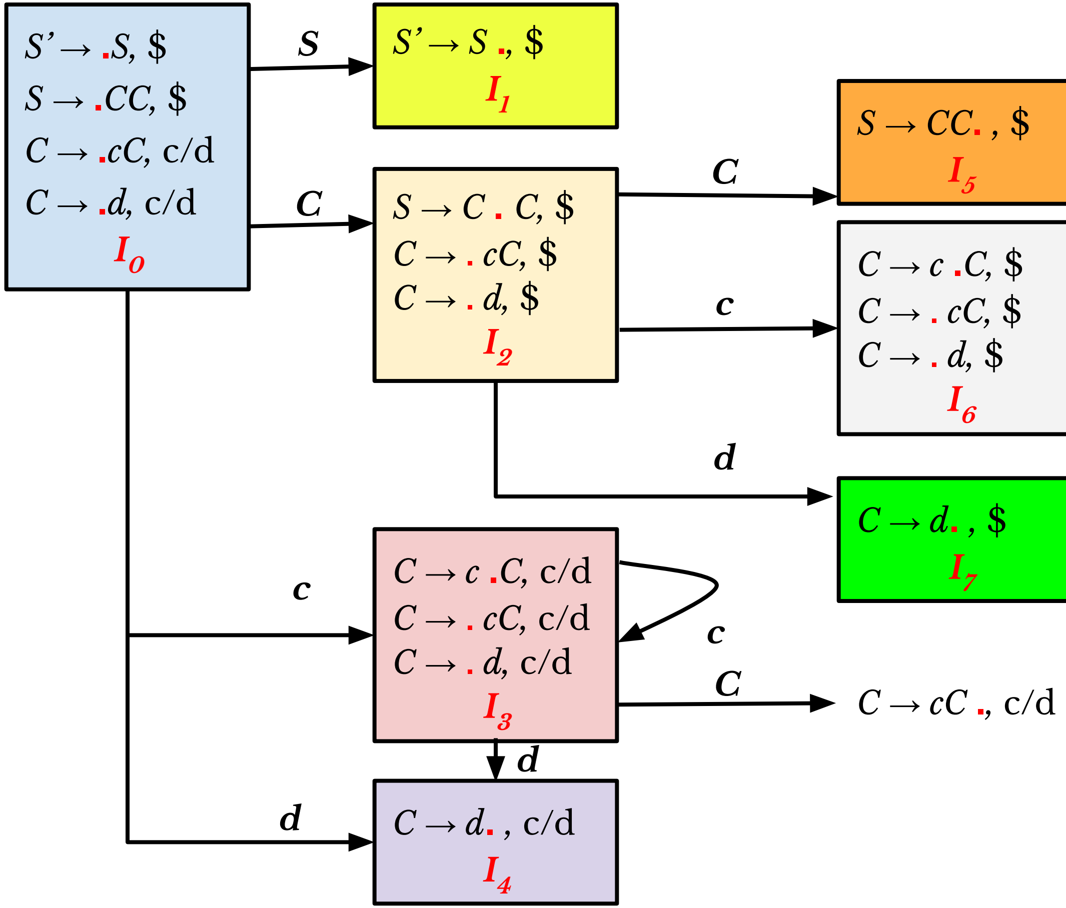
LR(1) Automation

0: $S' \to S$ **1:** $S \to CC$ **2:** $C \to c\,C$ **3:** $C \to d$



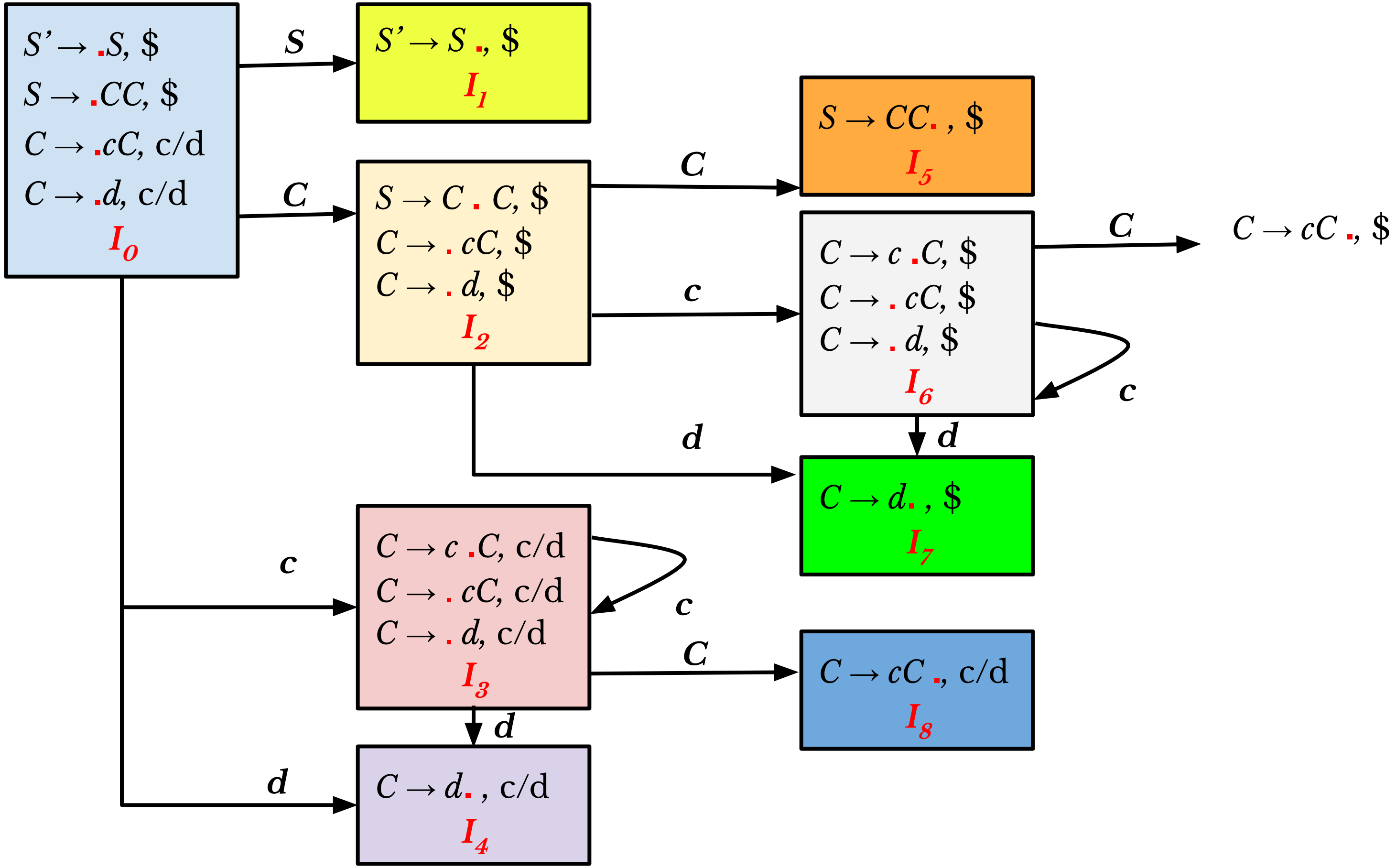
LR(1) Automation

0: $S' \rightarrow S$ 1: $S \rightarrow CC$ 2: $C \rightarrow c\ C$ 3: $C \rightarrow d$



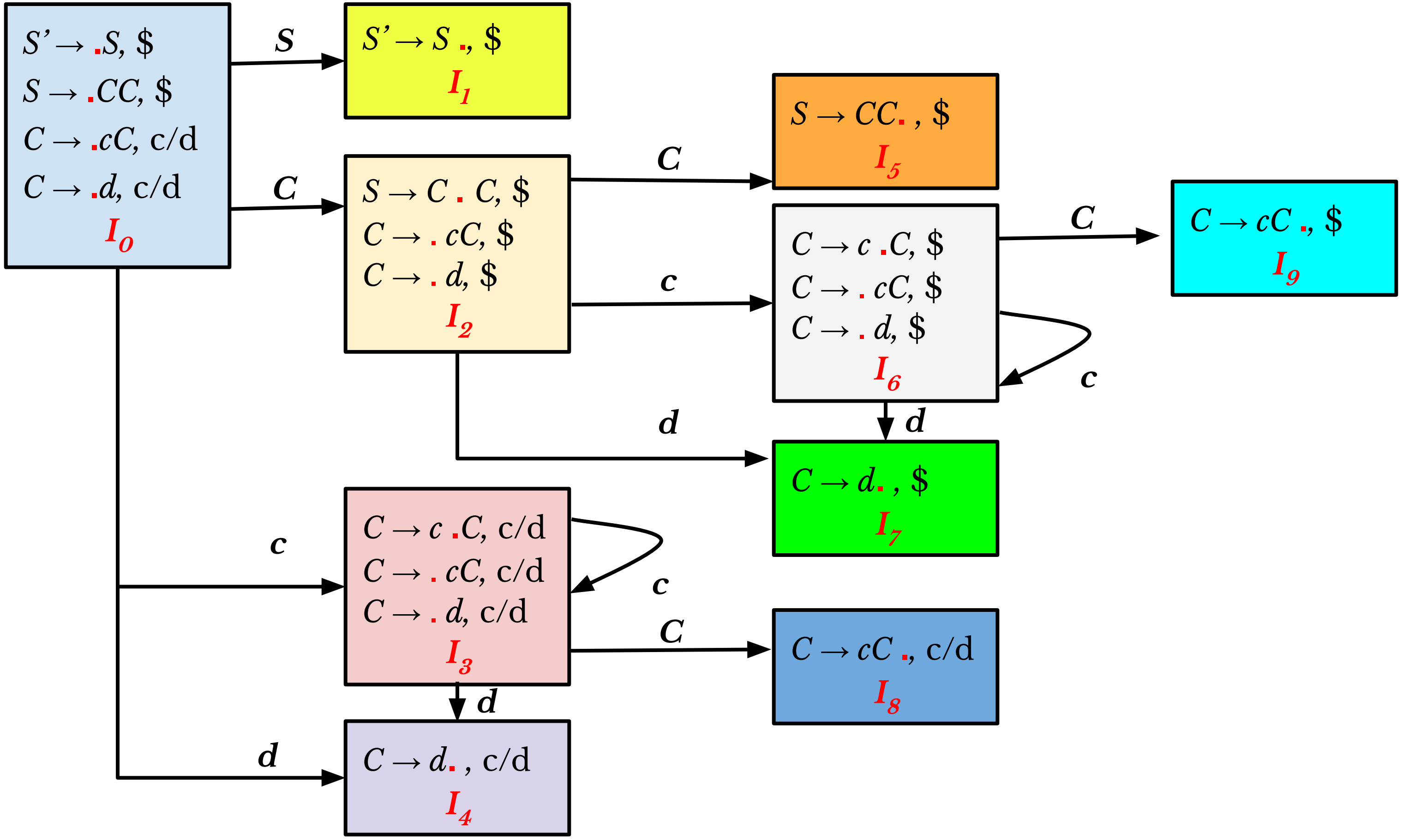
LR(1) Automation

0: $S' \to S$ **1:** $S \to CC$ **2:** $C \to c\ C$ **3:** $C \to d$



LR(1) Automation

0: $S' \to S$ **1:** $S \to CC$ **2:** $C \to c\,C$ **3:** $C \to d$



Building LR(1) parser: Parsing table

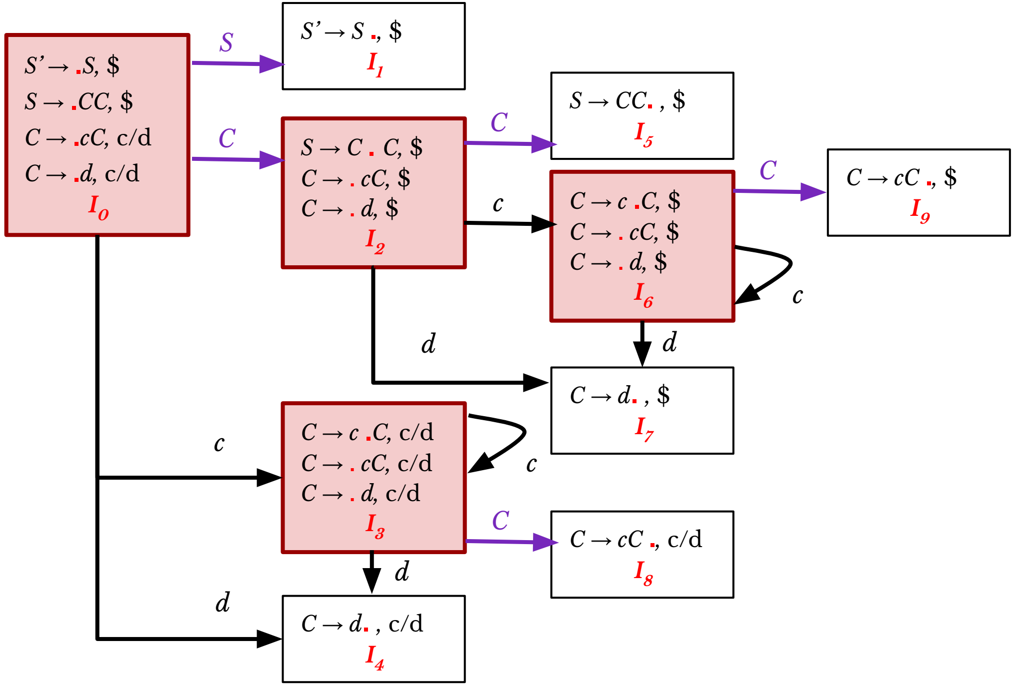
Constructing LR(1) parsing table

LR(1)-Table(G’)

1. Construct LR(1) collection for the grammar G’
2. Let I_i represents state S_i, then the parsing action for state i are as follows
   a. If $[A \rightarrow \alpha.a\beta,\ b]$ is in I_i and GOTO(I_i, a) = I_j
      i. Action[i, a] = “shift j”
   b. If $[A \rightarrow \alpha.,\ b]$ is in I_i
      i. Action[i, b] = “reduce $A \rightarrow \alpha$”
   c. If $[S' \rightarrow S.,\ \$]$ is in I_i
      i. Action[i, \$] = “accept”
3. For all non-terminals A,
   a. if GOTO(I_i, A) = I_j,
      i. GOTO[i, A] = j

LR(1) parsing table

0: $S' \to S$ **1:** $S \to CC$ **2:** $C \to c\,C$ **3:** $C \to d$



State	Action			GOTO	
	c	d	$	S	C
0				1	2
1					
2					5
3					8
4					
5					
6					9
7					
8					
9					

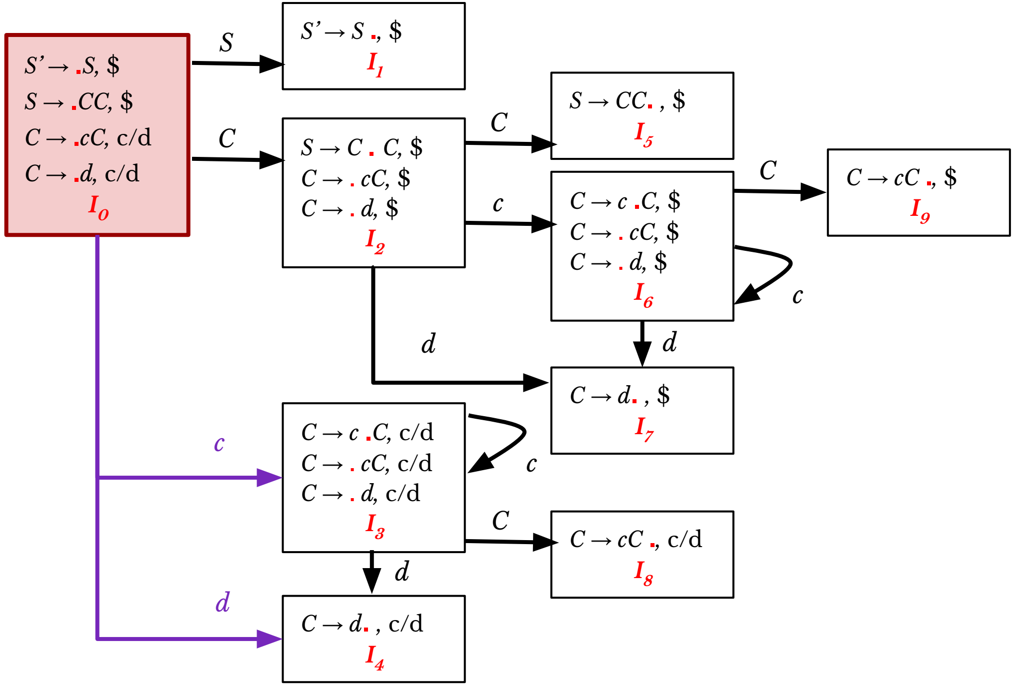
Rule 3: For all non-terminals A,

if GOTO(I_i, A) = I_j, then GOTO[i, A] = j

GOTO[0, S] = 1 GOTO[0, C] = 2

GOTO[2, C] = 5 GOTO[3, C] = 8 GOTO[6, C] = 9

LR(1) parsing table



Rule 2a: If $[A \to \alpha.a\beta,\ b]$ is in I_i and GOTO(I_i, a) = I_j

Action[i, a] = "shift j" or "s j"

Action[0, c] = s3

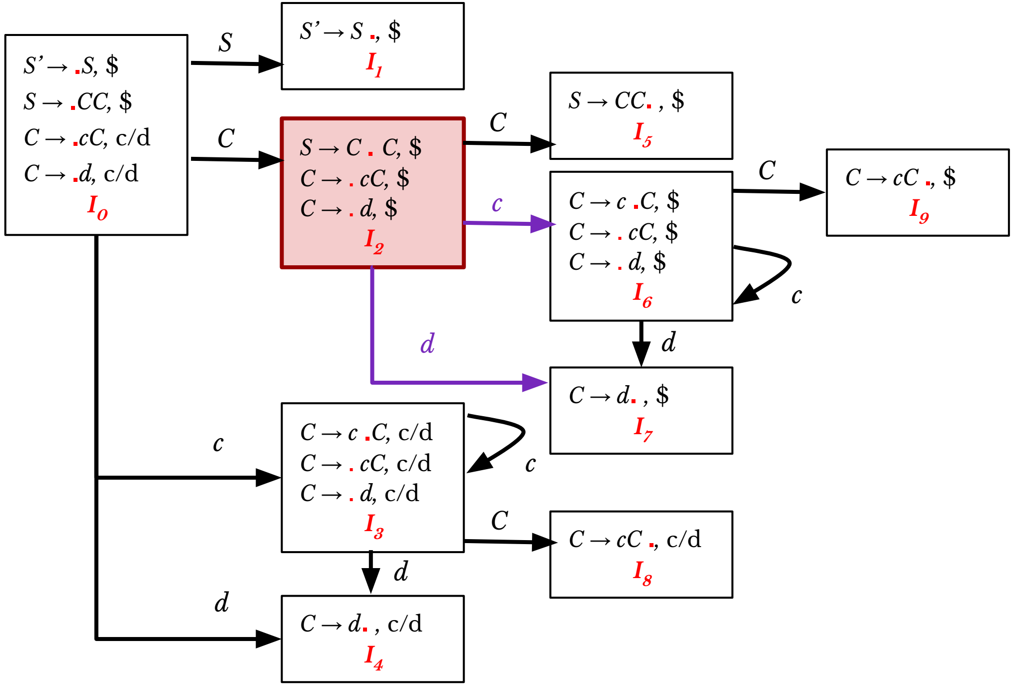
Action[0, d] = s4

0: $S' \to S$ **1:** $S \to CC$ **2:** $C \to c\,C$ **3:** $C \to d$

State	Action			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1					
2					5
3					8
4					
5					
6					9
7					
8					
9					

LR(1) parsing table

0: $S' \to S$ **1:** $S \to CC$ **2:** $C \to c\,C$ **3:** $C \to d$



State	Action: c	Action: d	Action: $	GOTO: S	GOTO: C
0	s3	s4		1	2
1					
2	s6	s7			5
3					8
4					
5					
6					9
7					
8					
9					

Rule 2a: If $[A \to \alpha.a\beta,\ b]$ is in I_i and GOTO$(I_i, a) = I_j$

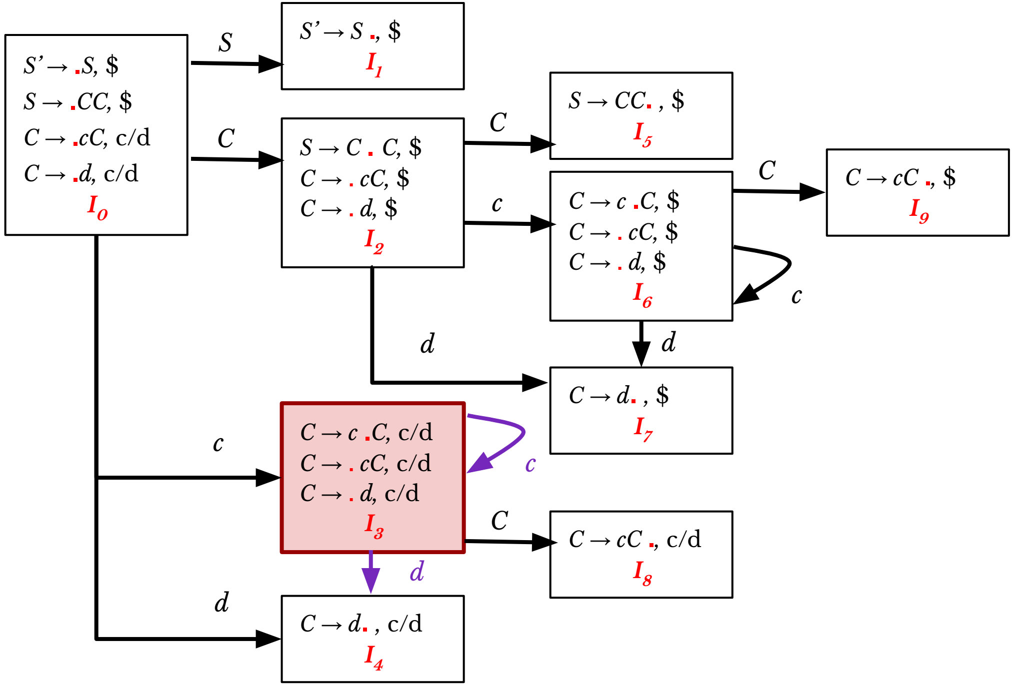
Action$[i, a]$ = "shift j" or "s j"

Action[2, c] = s6

Action[2, d] = s7

LR(1) parsing table

0: $S' \to S$ **1:** $S \to CC$ **2:** $C \to c\,C$ **3:** $C \to d$

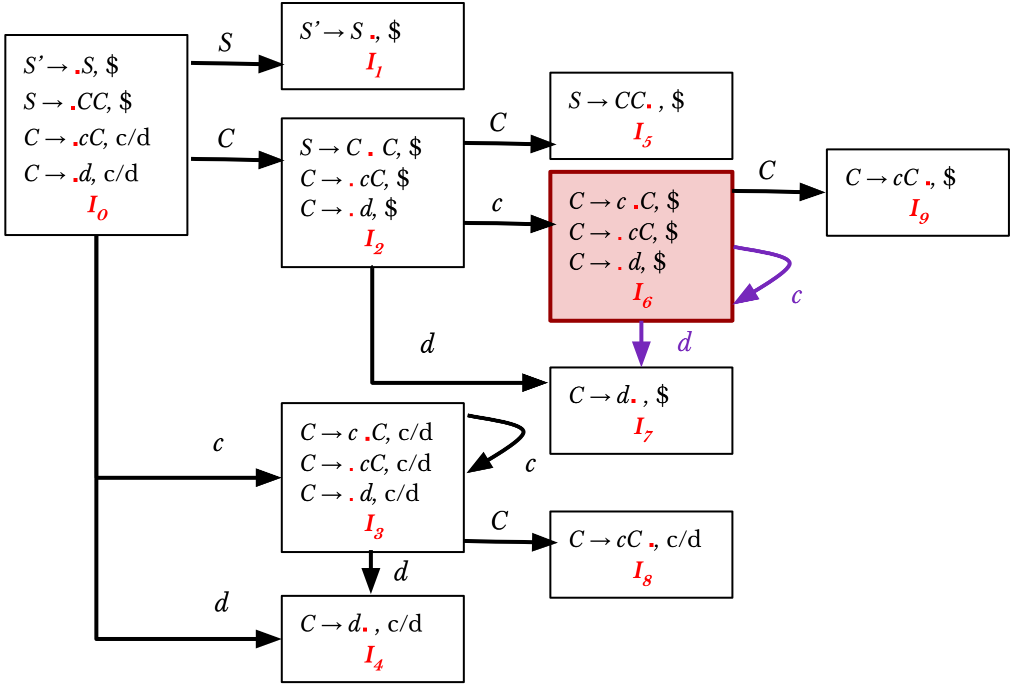


Rule 2a: If $[A \to \alpha.a\beta,\ b]$ is in I_i and GOTO$(I_i, a) = I_j$
Action$[i, a]$ = "shift j" or "s j"

Action[3, c] = s3
Action[3, d] = s4

State	Action			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1					
2	s6	s7			5
3	s3	s4			8
4					
5					
6					9
7					
8					
9					

LR(1) parsing table



Rule 2a: If $[A \to \alpha.a\beta,\ b]$ is in I_i and GOTO$(I_i, a) = I_j$

Action$[i, a]$ = "shift j" or "s j"

Action[6, c] = s6

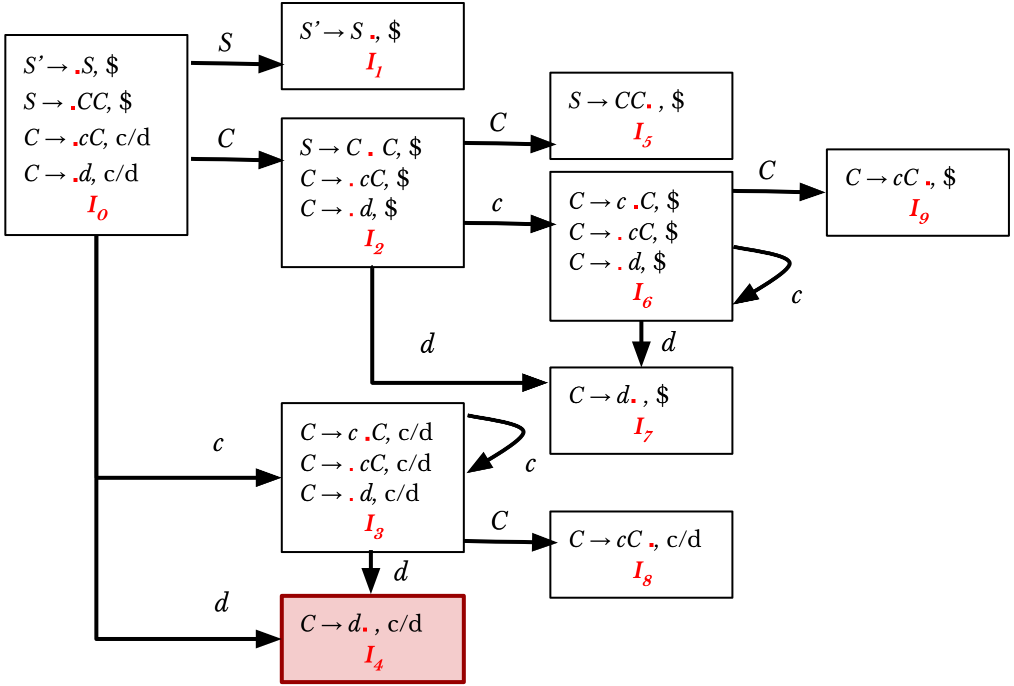
Action[6, d] = s7

0: $S' \to S$ 1: $S \to CC$ 2: $C \to c\,C$ 3: $C \to d$

State	Action			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1					
2	s6	s7			5
3	s3	s4			8
4					
5					
6	s6	s7			9
7					
8					
9					

LR(1) parsing table

0: $S' \to S$ **1:** $S \to CC$ **2:** $C \to c\,C$ **3:** $C \to d$



Rule 2b: If $[A \to \alpha., b]$ is in I_i

Action$[i, b]$ = "reduce $A \to \alpha$"

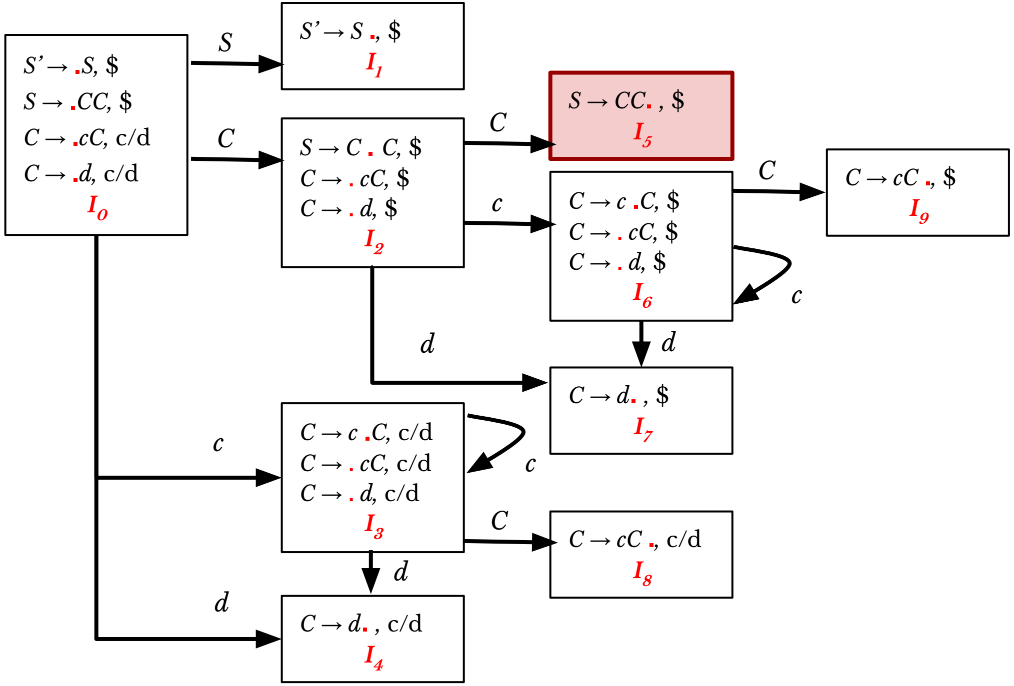
Action[4, c] = r3

Action[4, d] = r3

State	Action			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1					
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5					
6	s6	s7			9
7					
8					
9					

LR(1) parsing table

0: $S' \to S$ 1: $S \to CC$ 2: $C \to c\,C$ 3: $C \to d$



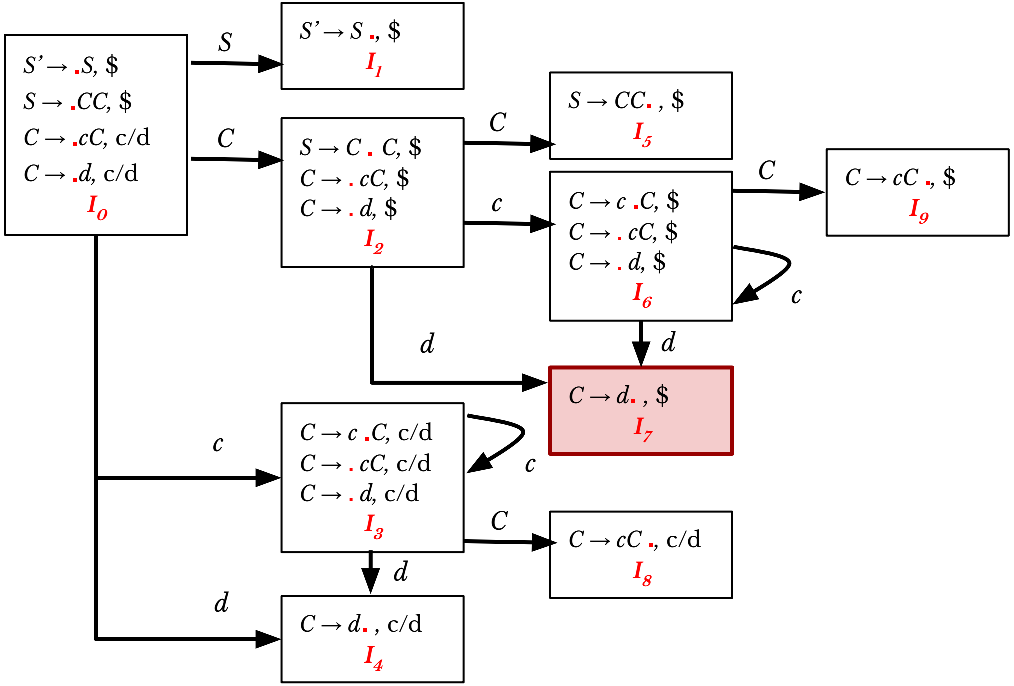
State	Action			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1					
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7					
8					
9					

Rule 2b: If $[A \to \alpha.,\ b]$ is in I_i

Action$[i,\ b]$ = "reduce $A \to \alpha$"

Action[5, $] = r1

LR(1) parsing table



Rule 2b: If $[A \to \alpha., b]$ is in I_i

Action[i , b] = "reduce $A \to \alpha$"

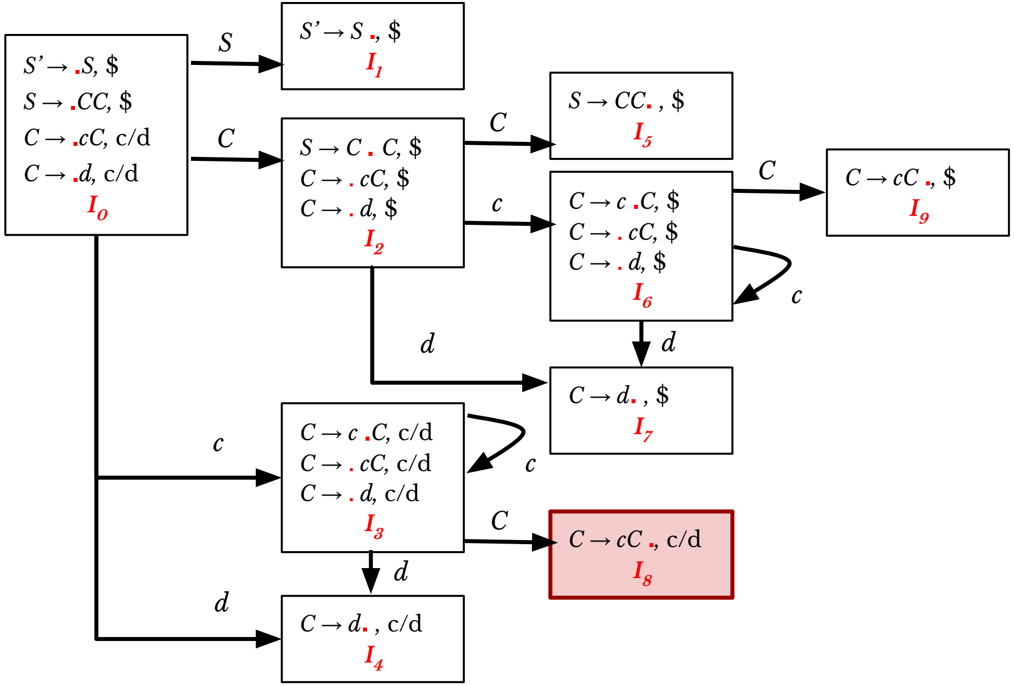
Action[7, $] = r3

0: $S' \to S$ **1:** $S \to CC$ **2:** $C \to c\,C$ **3:** $C \to d$

State	Action			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1					
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8					
9					

LR(1) parsing table

0: $S' \to S$ **1:** $S \to CC$ **2:** $C \to c\,C$ **3:** $C \to d$



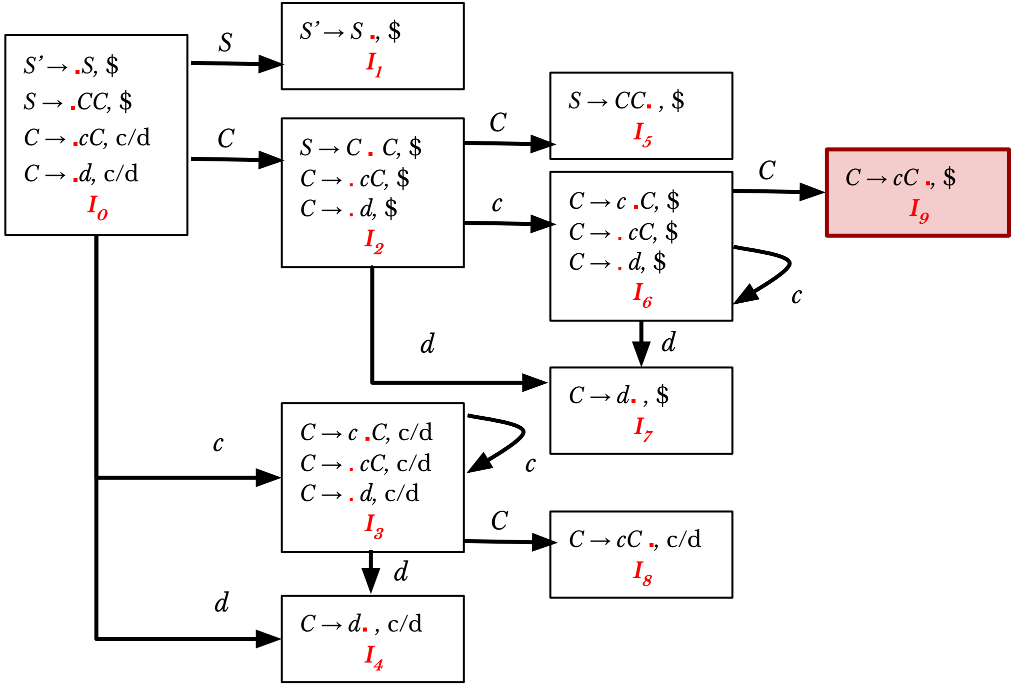
State	Action			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1					
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9					

Rule 2b: If $[A \to \alpha., b]$ is in I_i
Action$[i, b]$ = "reduce $A \to \alpha$"

Action[8, c] = r2
Action[8, d] = r2

LR(1) parsing table

0: $S' \to S$ 1: $S \to CC$ 2: $C \to c\,C$ 3: $C \to d$

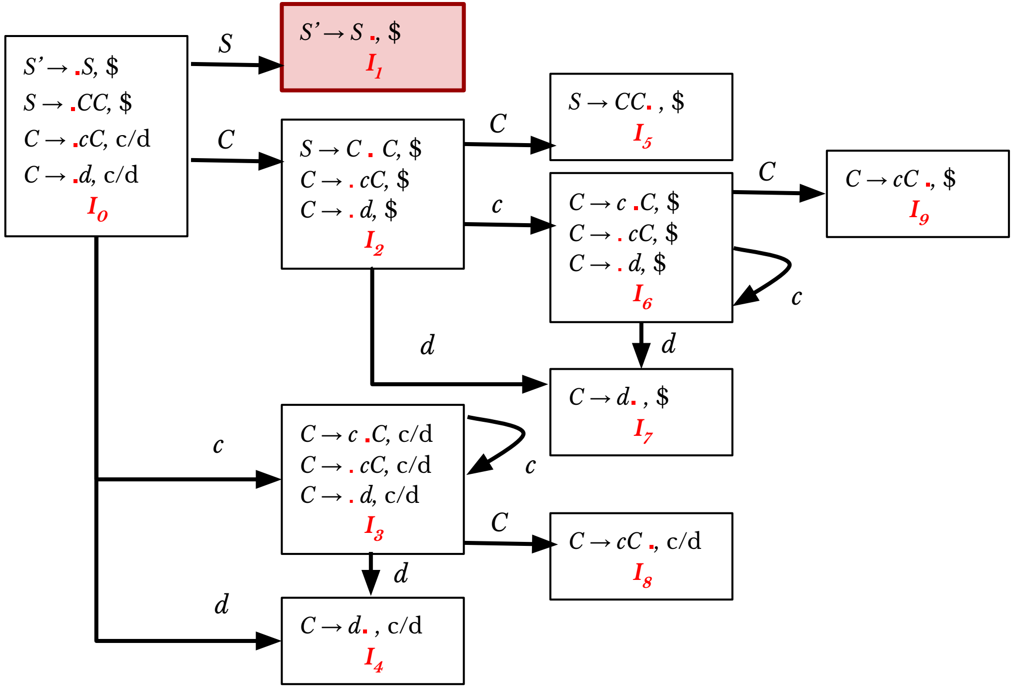


State	Action			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1					
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Rule 2b: If $[A \to \alpha., b]$ is in I_i
Action$[i, b]$ = "reduce $A \to \alpha$"

Action[9, $] = r2

LR(1) parsing table



Rule 2c: If $[S' \rightarrow S., \$]$ is in I_i
Action[i, $\$$] = "accept"

Action[1, $] = accept

0: $S' \rightarrow S$ **1:** $S \rightarrow CC$ **2:** $C \rightarrow c\,C$ **3:** $C \rightarrow d$

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		

All empty entries are "error" case.

LR(1) parser: Parsing

LR(1) parsing

- Parsing algorithm for all LR parsers are same.
- Exercise:
    - Input: “cccdddd”

LALR parsing

- Often used in practice
- Most common syntactic constructs of programming languages can be expressed conveniently by an LALR.
- SLR and LALR tables always have the same number of states
  - Roughly, several hundred states for the C language
- Table size is considerably small than canonical LR or LR(1)
  - Canonical LR has, roughly, several thousand states for the same language.

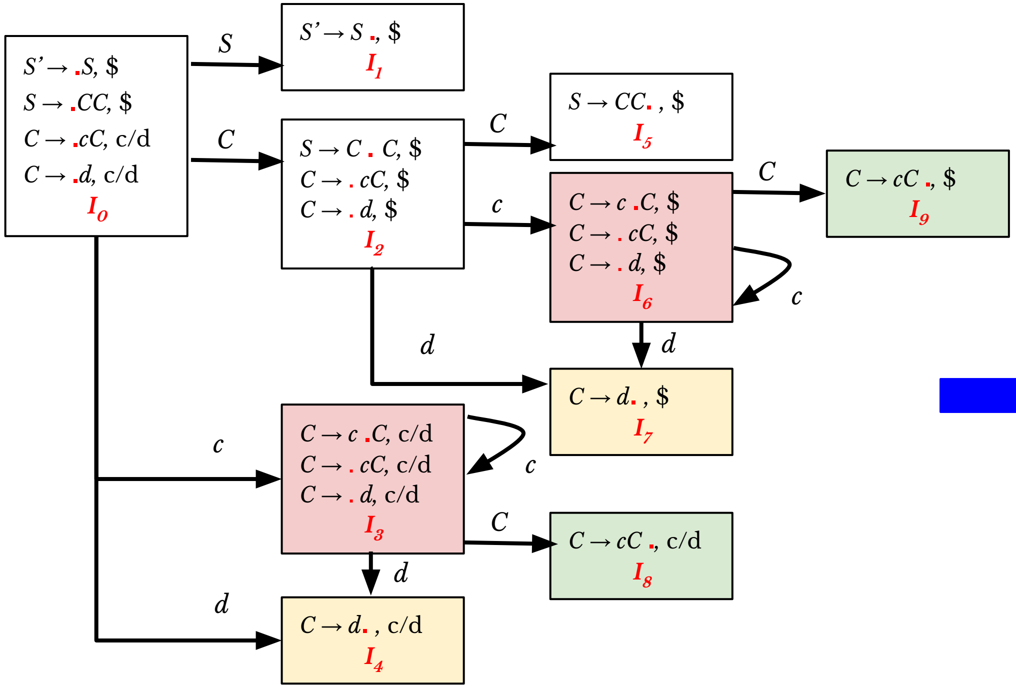
For item $[A \to \alpha.a\beta,\ \ b]$
Core: $A \to \alpha.a\beta$
Lookahead: b

Constructing LALR parsers

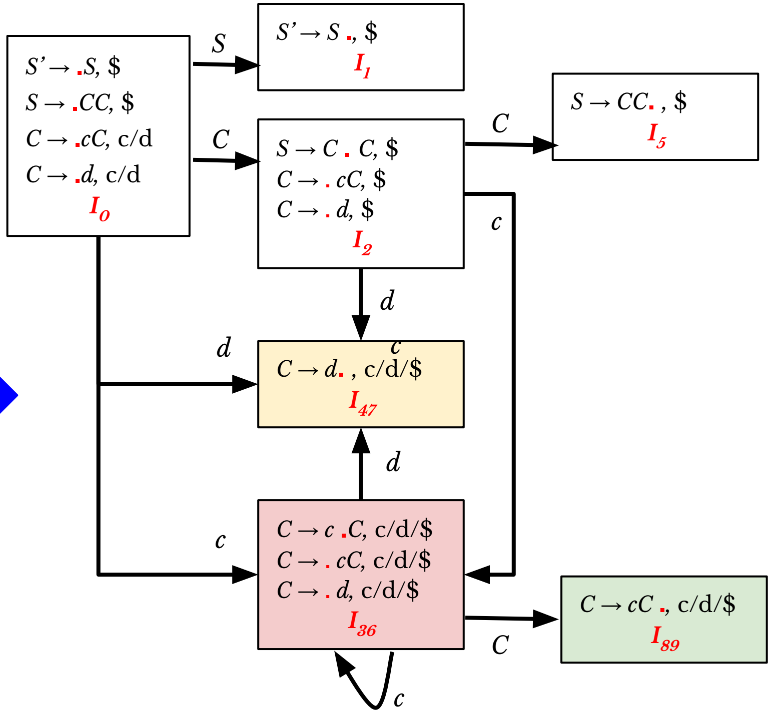
- Canonical set of items for LALR automation
  - Look for the item sets I_i and I_j in LR(1) automation, such that
    - Cores of Item(I_i) == Cores of Item(I_j), with different lookahead symbols
      - $I_4 = [C \to d. ,\ \mathrm{c/d}]$
      - $I_7 = [C \to d. ,\ \$]$
  - Merge the item sets I_i and I_j into I_{ij}, such that
    - Item set I_{ij} contains all items, with lookahead symbols merged
      - $I_{47} = [C \to d. ,\ \mathrm{c/d/\$}]$

LALR Automation

0: $S' \rightarrow S$ **1:** $S \rightarrow CC$ **2:** $C \rightarrow c\,C$ **3:** $C \rightarrow d$



LR Automation



LALR Automation

LALR parsing table

0: $S' \rightarrow S$ **1:** $S \rightarrow CC$ **2:** $C \rightarrow c\, C$ **3:** $C \rightarrow d$

State	Action			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

LALR grammar

- Its possible to introduce reduce/reduce conflicts during merger

  $I_1 = \{[A \to \alpha. , a], [B \to \beta. , b]\}$ $\qquad$ $I_2 = \{[A \to \alpha. , b], [B \to \beta. , c]\}$

  $\Rightarrow \quad I_{12} = \{[A \to \alpha. , a/b], [B \to \beta. , b/c]\}$

  Action(12, b) = reduce with A or B??

- Cannot introduce a shift/reduce conflict
  - Suppose the merged item introduced a shift/reduce conflict, e.g., on symbol a

    $I_{34} = \{[A \to \alpha. , a/b], [B \to \beta.a\,\gamma , b/c]\}$ Action(34, a) = shift / reduce?
  - This means that we had two items in the LR(1) set as

    $I_3 = \{[A \to \alpha. , a], [B \to \beta.a\,\gamma , b]\}$ $\qquad$ $I_4 = \{[A \to \alpha. , b], [B \to \beta.a\,\gamma , c]\}$
  - Observe, there is a shift/reduce conflict prior to the merge operation, i.e., the original grammar was not LR(1).

- Grammar is LALR(1), if no conflicts are introduced

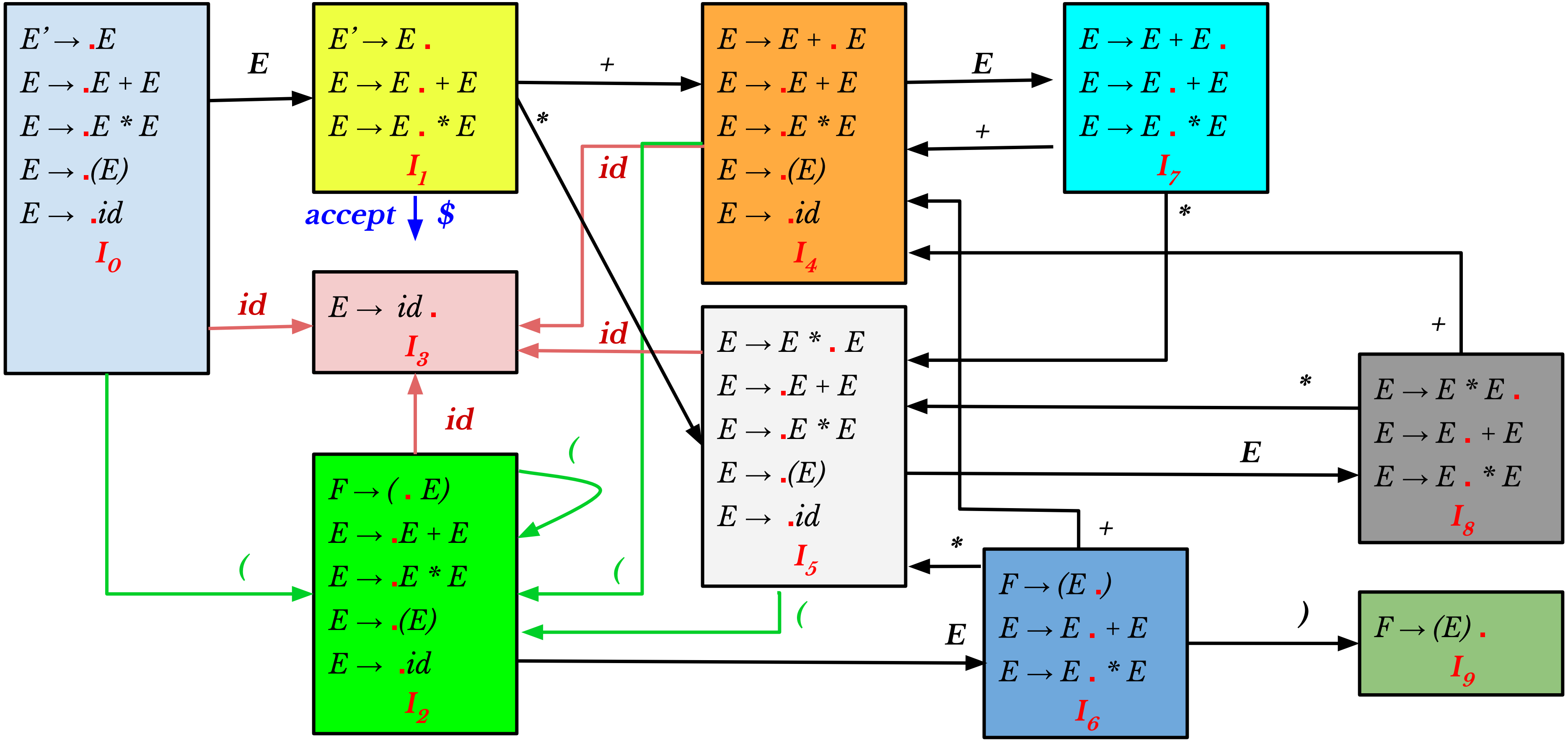
Parser and Ambiguous grammar

LR parsers for Ambiguous grammars

- Grammars for the construction of LR-parsing tables must be unambiguous
- *Can we create LR-parsing tables for ambiguous grammars?*
  - Yes, but they will have conflicts
  - What if, we can resolve these conflicts in favor of one of them to disambiguate the grammar?
    - At the end, we will have again an unambiguous grammar
- *Why we want to use an ambiguous grammar?*
  - Some of the ambiguous grammars are much natural, and a corresponding unambiguous grammar can be very complex
  - Usage of an ambiguous grammar may eliminate unnecessary reductions
    - $E \Rightarrow T \Rightarrow F \Rightarrow id$ (using unambiguous expression-grammar)
    - $E \Rightarrow id$ (using ambiguous expression-grammar)

SLR(1) Automation

0: $E' \to E$ 1: $E \to E+E$ 2: $E \to E*E$ 3: $E \to (E)$
4: $E \to id$



SLR parsing table

State	Action						GOTO
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			**acc**	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		s4/r1	s5/r1		r1	r1	
8		s4/r2	s5/r2		r2	r2	
9		r3	r3		r3	r3	

LR parsers for Ambiguous grammars

Why ambiguity is a problem?
We have a decision to make and not sure which parse tree to pick?

- Ambiguous grammars G: $E \rightarrow E + E \mid E * E \mid (E) \mid id$
- We can have two parse trees for an input
  - Input 1: id + id + id
    P1: $E \Rightarrow E + E \Rightarrow id + E \Rightarrow id + E + E \Rightarrow id + id + E \Rightarrow id + id + id$
    P2: $E \Rightarrow E + E \Rightarrow E + E + E \Rightarrow id + E + E \Rightarrow id + id + E \Rightarrow id + id + id$
  - Input 2: id * id * id
    P1: $E \Rightarrow E * E \Rightarrow id * E \Rightarrow id * E * E \Rightarrow id * id * E \Rightarrow id * id * id$
    P2: $E \Rightarrow E * E \Rightarrow E * E * E \Rightarrow id * E * E \Rightarrow id * id * E \Rightarrow id * id * id$
  - Input 3: id + id * id
    P1: $E \Rightarrow E + E \Rightarrow id + E \Rightarrow id + E * E \Rightarrow id + id * E \Rightarrow id + id * id$
    P2: $E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow id + E * E \Rightarrow id + id * E \Rightarrow id + id * id$
  - Input 4: id * id + id
    P1: $E \Rightarrow E * E \Rightarrow id * E \Rightarrow id * E + E \Rightarrow id * id + E \Rightarrow id * id + id$
    P2: $E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow id * E + E \Rightarrow id * id + E \Rightarrow id * id + id$

Computation-wise, which parse tree is correct?
Input 1:
Either P1 or P2
Input 2:
Either P1 or P2
Input 3:
P1
Input 4:
P2

Reason?
Operation ‘*’ has precedence over operator ‘+’

LR parsers for Ambiguous grammars

- In the parsing-table of an ambiguous grammars, if we can explicitly resolve the conflicts, then the processing of parsing remains unambiguous
  - E.g.,
    - We can look for the precedence of operators for the conflict resolution, 'OR'
    - We can look for the associativity of operators for the conflict resolution

In state 7, we have shift/reduce conflicts for symbols + and *

$$I_0 \overset{E}{\Rightarrow} I_1 \overset{+}{\Rightarrow} I_4 \overset{E}{\Rightarrow} I_7$$

If current input symbol is +

Shift → if + right associative

Reduce → if + left associative

If current input symbol is *

Shift → if * has higher precedence over +

Reduce→ if + has higher precedence over *

In state 8, we have shift/reduce conflicts for symbols + and *

$$I_0 \overset{E}{\Rightarrow} I_1 \overset{*}{\Rightarrow} I_4 \overset{E}{\Rightarrow} I_7$$

If current input symbol is *

Shift → if * right associative

Reduce → if * left associative

If current input symbol is +

Shift → if + has higher precedence over *

Reduce→ if * has higher precedence over +

SLR parsing table

Conflict-free parsing table for ambiguous grammar.

State	Action						GOTO
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			**acc**	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	