Syntax Analysis

Md Shad Akhtar
Assistant Professor
IIIT Dharwad

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.

G: S → aABe
A → Abc | b
B → d
Input: abbcde

Bottom-Up parsing: An example

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

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 $E \to T$ or $F \to id_2$?

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

Input: $id_1 * id_2$

Stack	Input	Action
$	$id_1 * id_2$ $	Shift
$ **id_1**	$* id_2$ $	Reduce by F → id
$ **F**	$* id_2$ $	Reduce by T → F
$ T	$* id_2$ $	Shift
$ T *	id_2 $	Shift
$ T * **id_2**	$	Reduce by F → id
$ **T * F**	$	Reduce by T → T*F
$ **T**	$	Reduce by E → 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

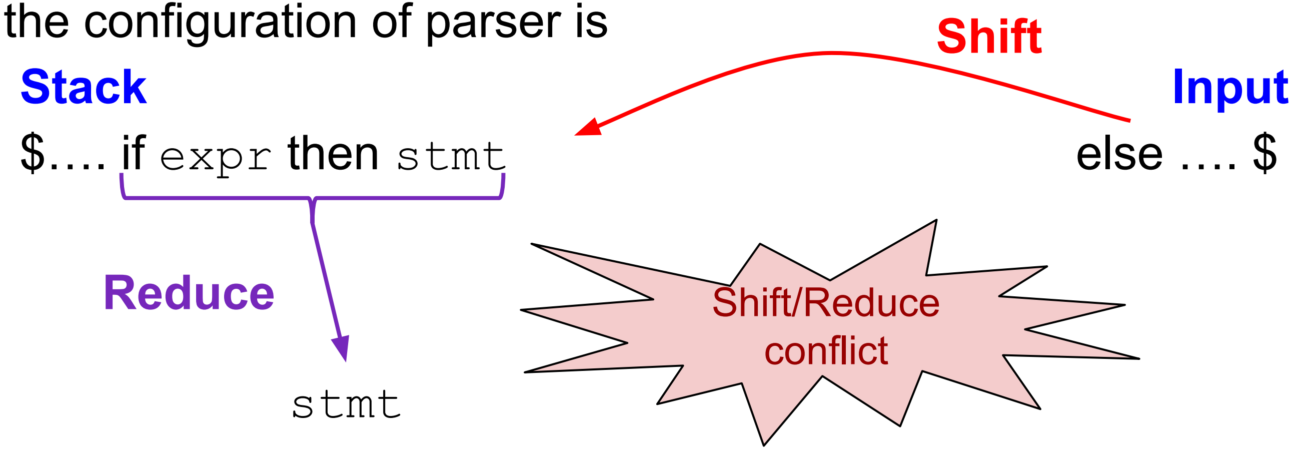
- There are grammars for which shift-reduce parsing cannot be used.
- Shift-Reduce parser for such grammars may have a configuration where the parser cannot decide whether to
    - Shift the symbols onto the stack or Reduce the handle to a non-terminal OR
    - Reduce the handle with some non-terminal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

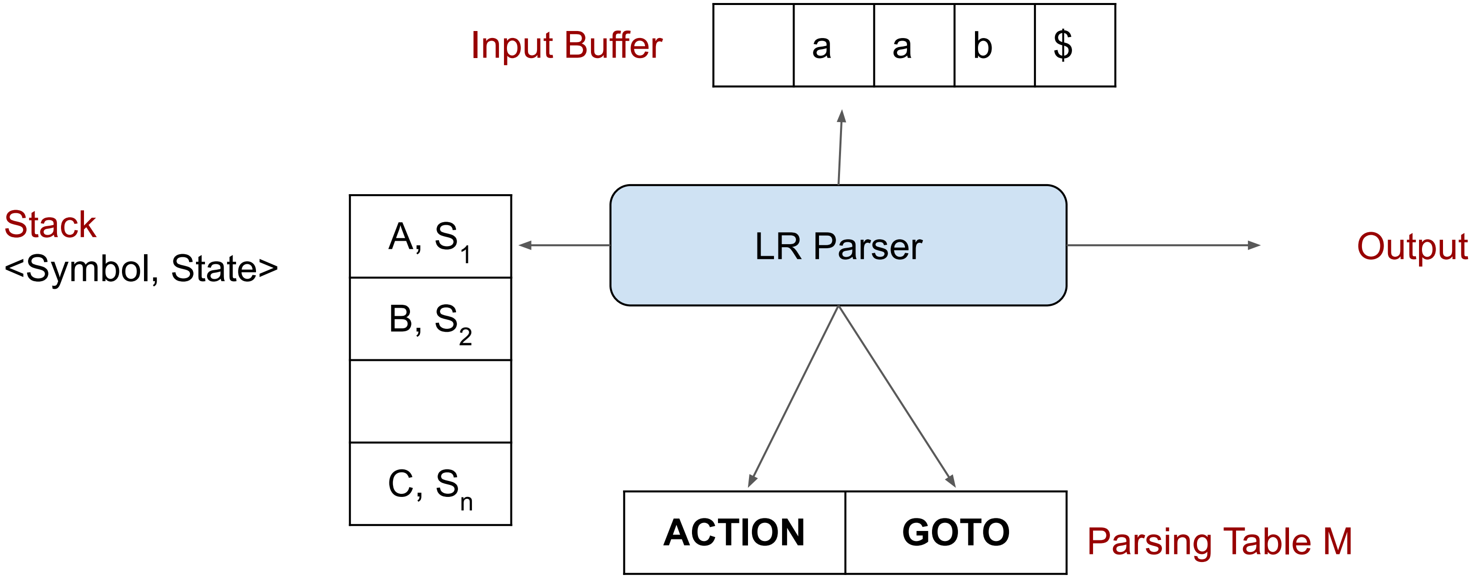
- Three main types of parse
  - Simple LR or SLR or LR(0)
  - Canonical LR or LR(1)
  - Look-ahead LR or LALR
- Parsing of all three parsers are similar, only their parsing tables are different

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

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

  - **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]

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

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 $A \rightarrow \alpha.B\beta$ is in Closure(I) and $B \rightarrow \gamma$ is a production
   a. Add item $B \rightarrow .\gamma$ 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 $A \rightarrow \alpha.B\beta$
  - GOTO (I, B) = Closure($A \rightarrow \alpha B.\beta$)

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

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 $S' \rightarrow S$, if S was the starting symbol.
    - S' becomes new start symbol
    - $S' \rightarrow S.$ signifies the acceptance of the input.

Computation of the canonical LR(0) collection

```
Items($G'$)
1. $C$ = Closure($\{[S' \to .S]\}$)
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:** $F \rightarrow id$

$E' \rightarrow .E$

$E \rightarrow .E + T$

$E \rightarrow .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

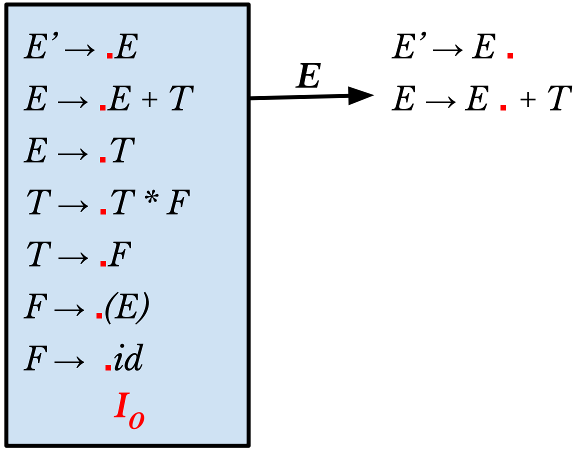
$F \rightarrow .(E)$

$F \rightarrow .id$

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

E

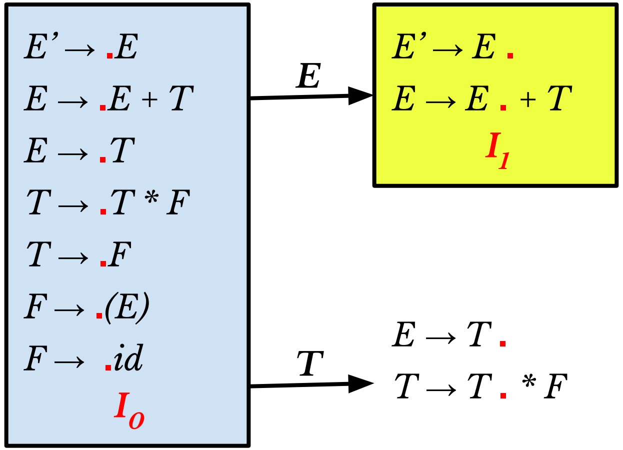
$E' \rightarrow E\,.$

$E \rightarrow E\,. + T$

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:** $F \to id$



$E' \to .E$

$E \to .E + T$

$E \to .T$

$T \to .T * F$

$T \to .F$

$F \to .(E)$

$F \to .id$

I_0

E

$E' \to E .$

$E \to E . + T$

I_1

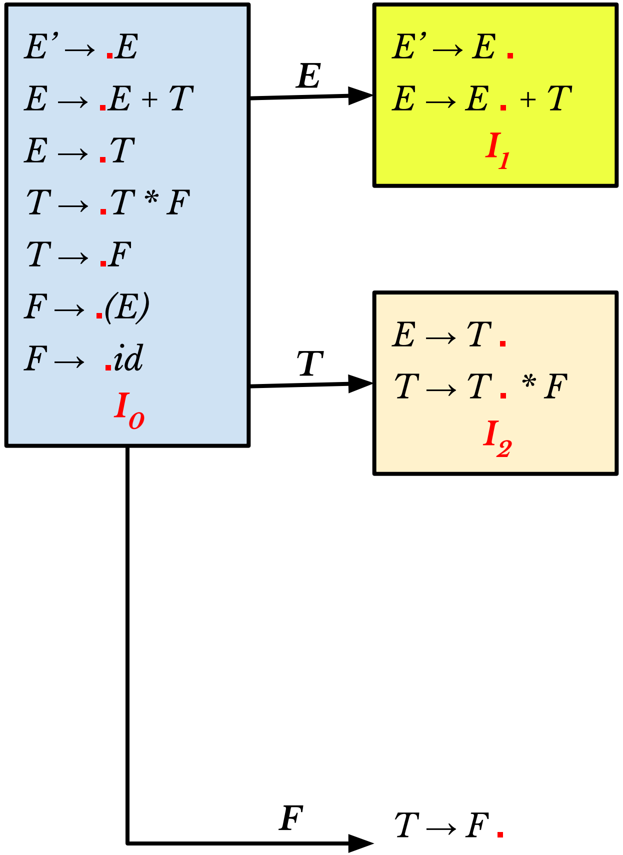
T

$E \to T .$

$T \to T . * F$

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:** $F \rightarrow id$



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

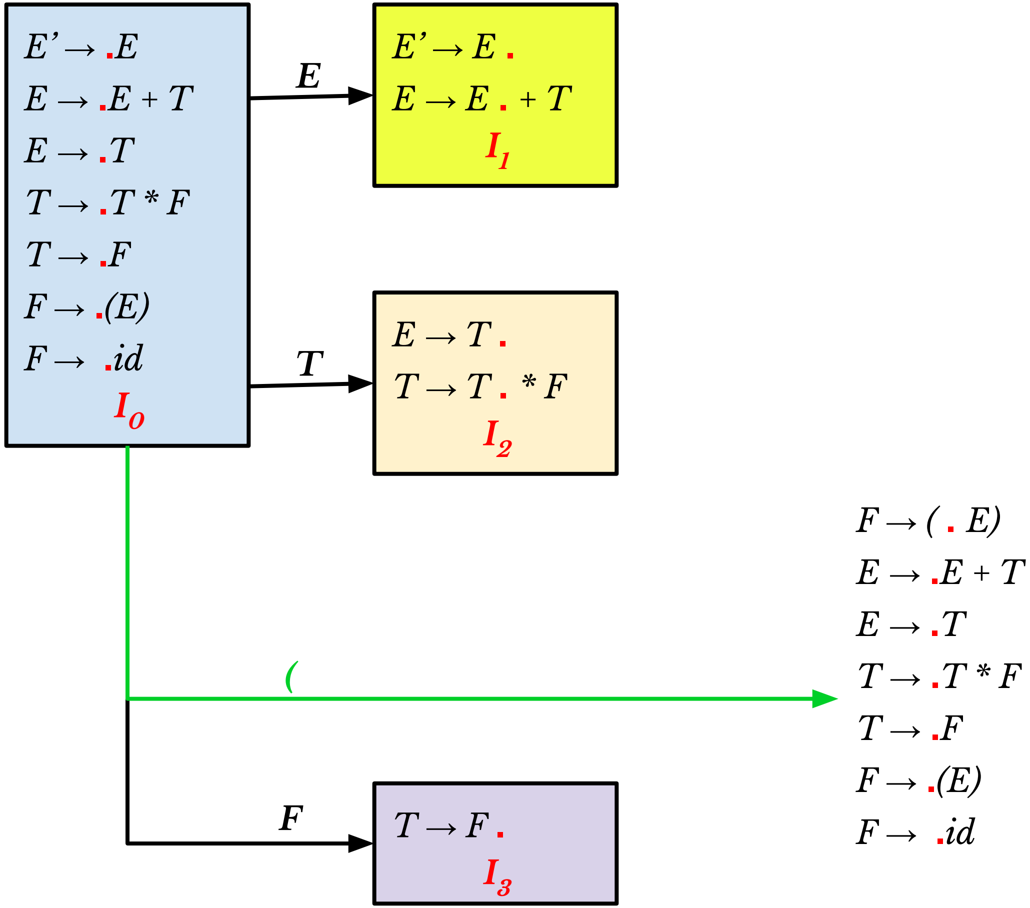
$E' \rightarrow E\,.$
$E \rightarrow E\,. + T$
I_1

$E \rightarrow T\,.$
$T \rightarrow T\,. * F$
I_2

$T \rightarrow F\,.$

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: $F \to id$



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

$E' \to E .$
$E \to E . + T$
I_1

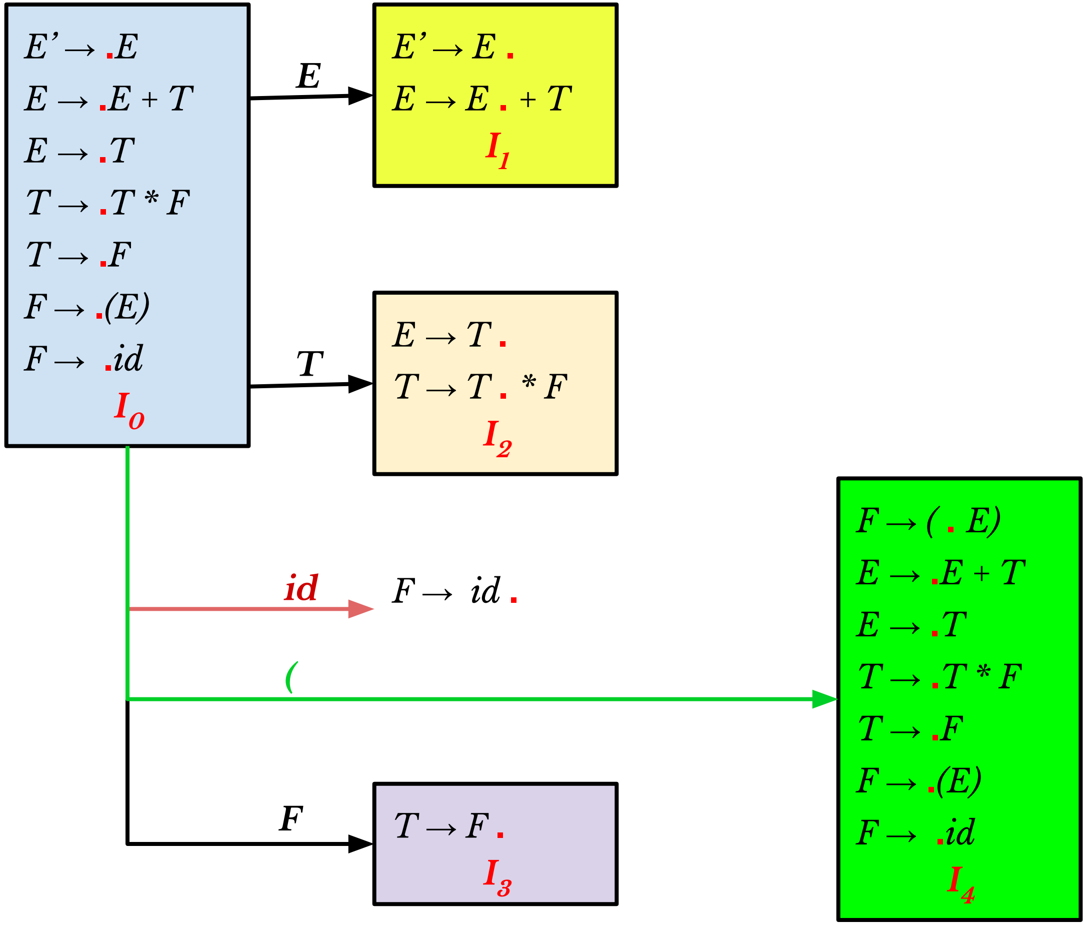
$E \to T .$
$T \to T . * F$
I_2

$T \to F .$
I_3

$F \to (. E)$
$E \to .E + T$
$E \to .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .id$

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: $F \to id$



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

$E' \to E.$
$E \to E. + T$
I_1

$E \to T.$
$T \to T. * F$
I_2

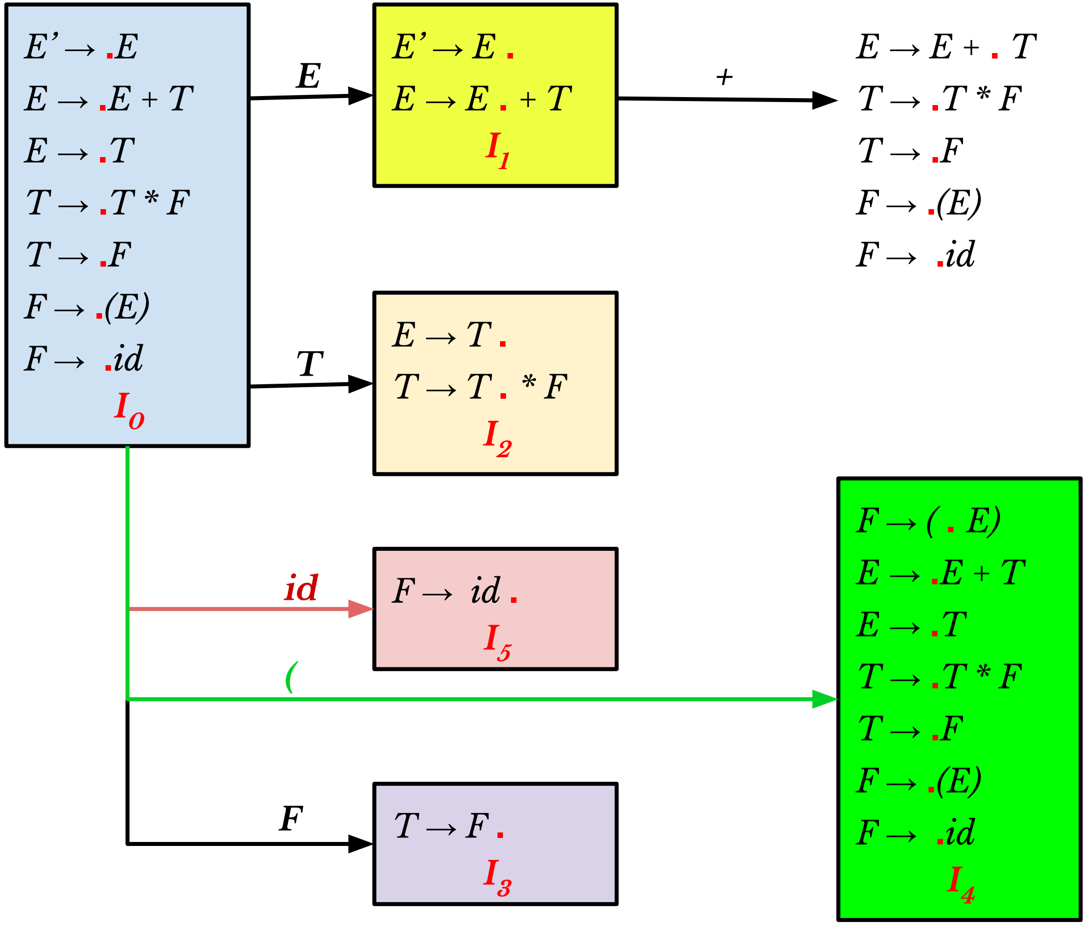
$F \to id.$

$T \to F.$
I_3

$F \to (.E)$
$E \to .E + T$
$E \to .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .id$
I_4

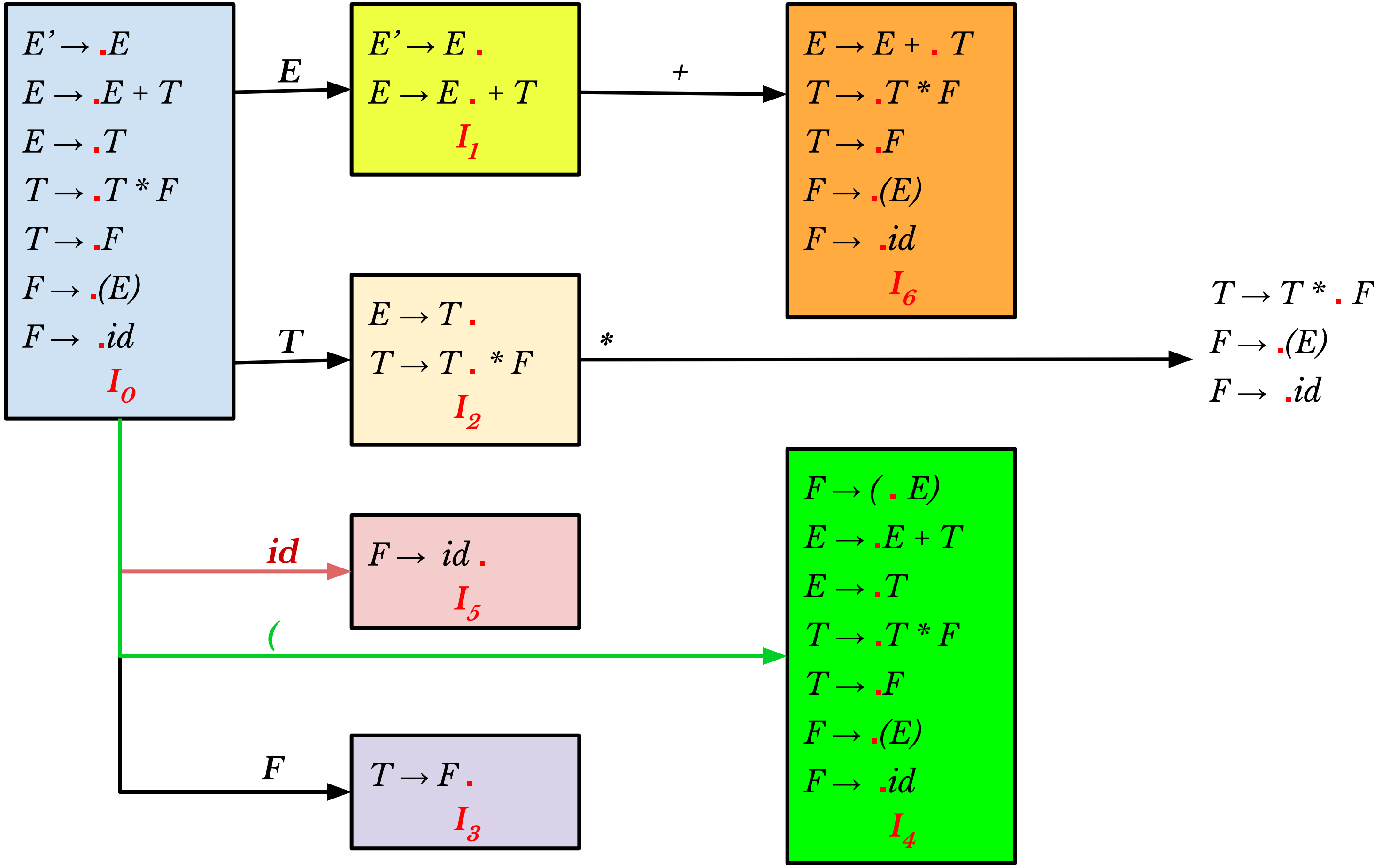
LR(0) Automation

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:** $F \to id$



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

$E' \to E .$
$E \to E . + T$
I_1

$E \to E + . T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .id$
I_6

$E \to T .$
$T \to T . * F$
I_2

$T \to T * . F$
$F \to .(E)$
$F \to .id$

$F \to id .$
I_5

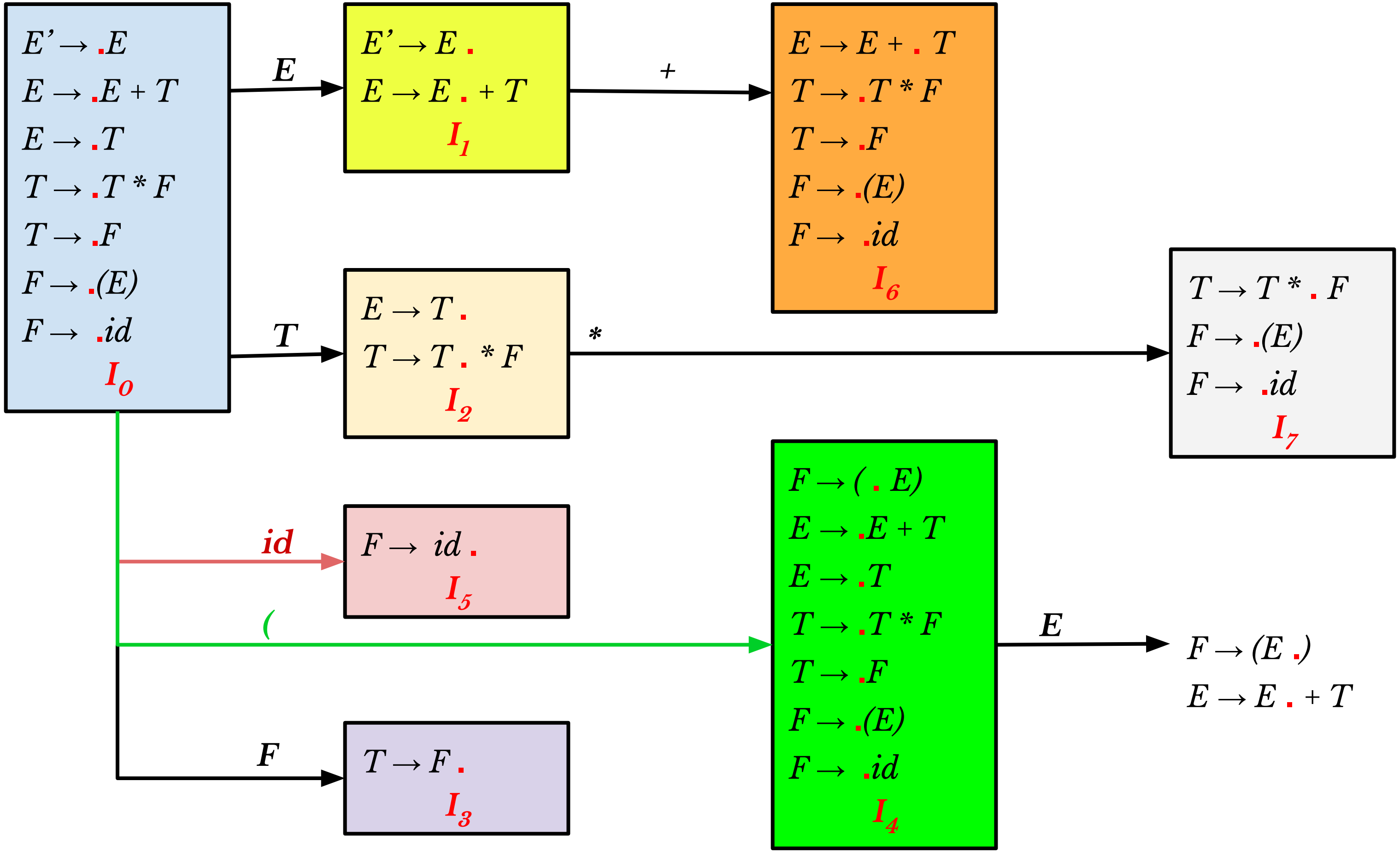
$F \to (. E)$
$E \to .E + T$
$E \to .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .id$
I_4

$T \to F .$
I_3

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:** $F \to id$



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

$E' \to E .$
$E \to E . + T$
I_1

$E \to T .$
$T \to T . * F$
I_2

$T \to F .$
I_3

$F \to (. E)$
$E \to .E + T$
$E \to .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .id$
I_4

$F \to id .$
I_5

$E \to E + . T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .id$
I_6

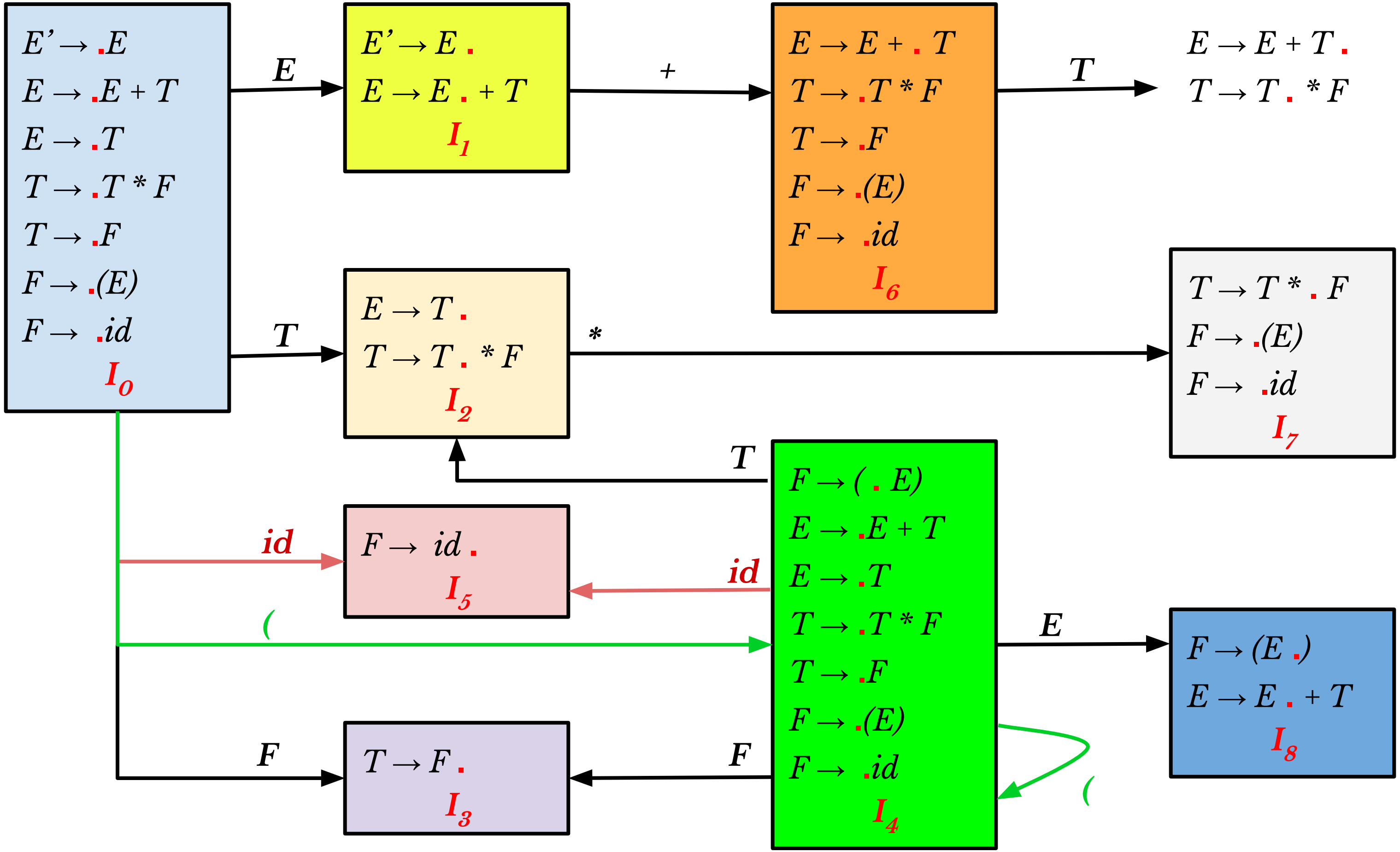
$T \to T * . F$
$F \to .(E)$
$F \to .id$
I_7

$F \to (E .)$
$E \to E . + T$

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_1:
$E' \to E .$
$E \to E . + T$

I_2:
$E \to T .$
$T \to T . * F$

I_3:
$T \to F .$

I_4:
$F \to (. E)$
$E \to .E + T$
$E \to .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_5:
$F \to id .$

I_6:
$E \to E + . T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .id$

$E \to E + T .$
$T \to T . * F$

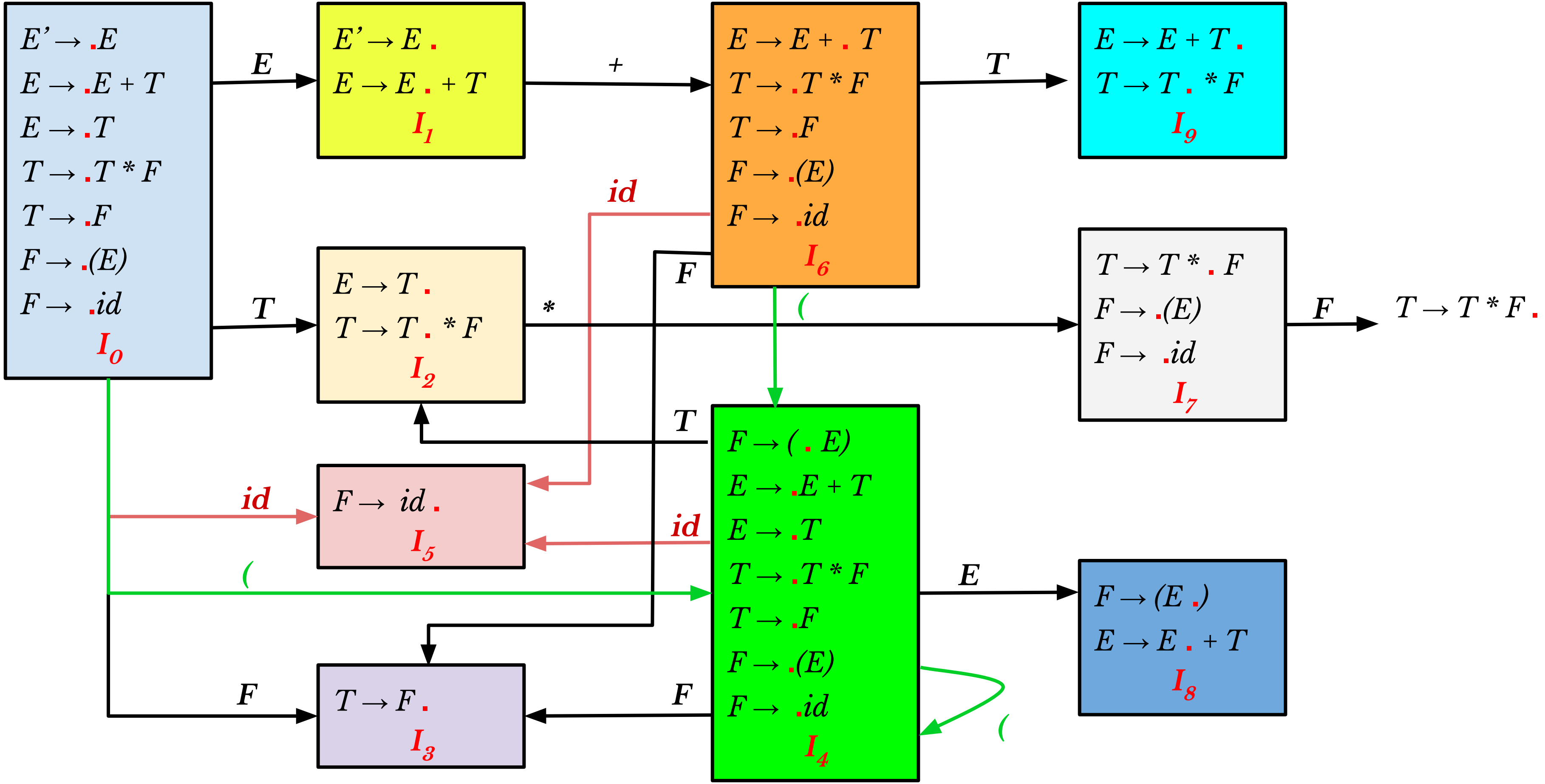
I_7:
$T \to T * . F$
$F \to .(E)$
$F \to .id$

I_8:
$F \to (E .)$
$E \to E . + T$

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

$E' \rightarrow E .$
$E \rightarrow E . + T$
I_1

$E \rightarrow E + . T$
$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$
I_6

$E \rightarrow E + T .$
$T \rightarrow T . * F$
I_9

$E \rightarrow T .$
$T \rightarrow T . * F$
I_2

$T \rightarrow T * . F$
$F \rightarrow .(E)$
$F \rightarrow .id$
I_7

$T \rightarrow T * F .$

$F \rightarrow id .$
I_5

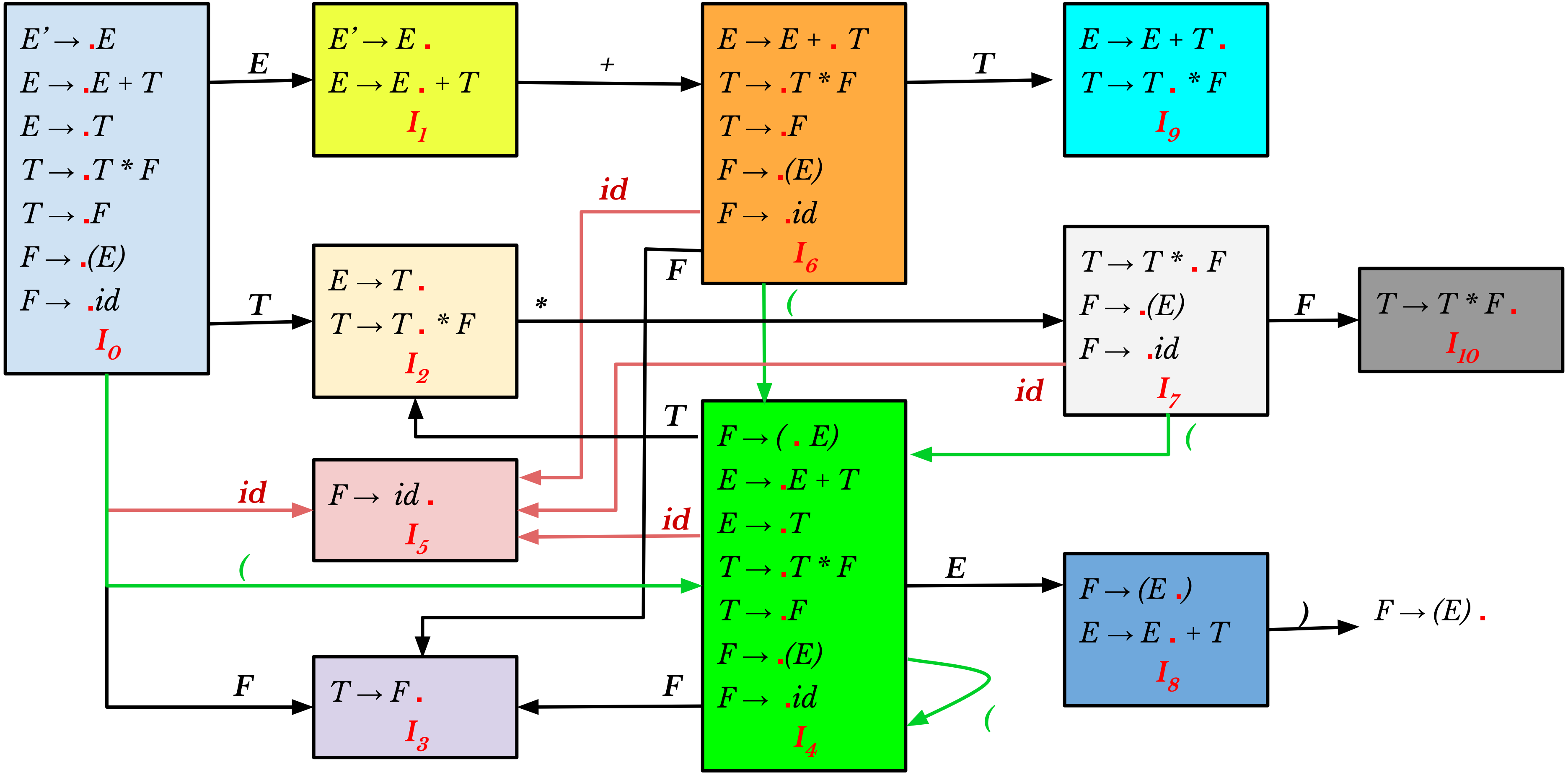
$F \rightarrow (. E)$
$E \rightarrow .E + T$
$E \rightarrow .T$
$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$
I_4

$F \rightarrow (E .)$
$E \rightarrow E . + T$
I_8

$T \rightarrow F .$
I_3

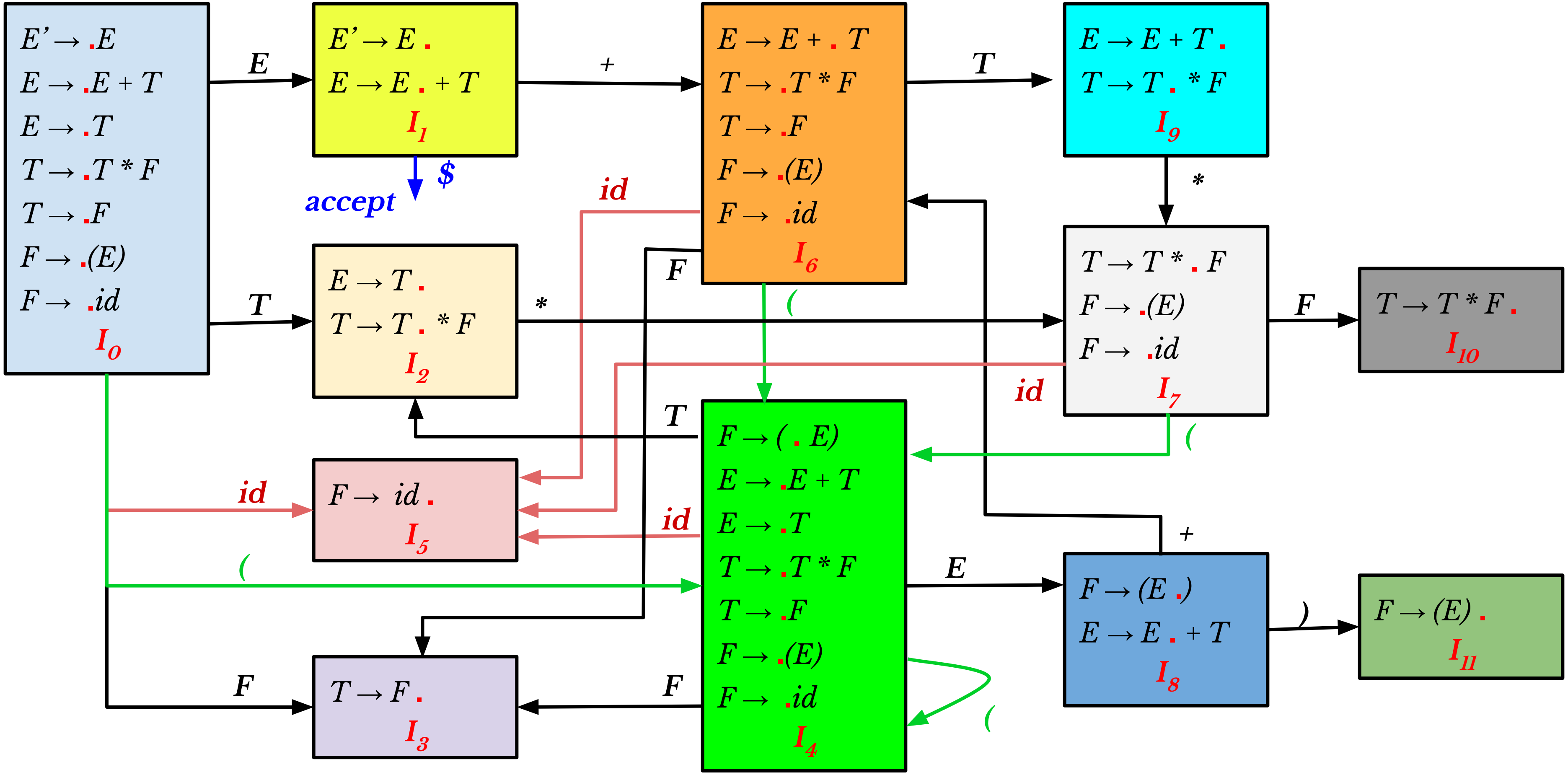
LR(0) Automation

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: $F \to id$



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

$E' \to E\ .$
$E \to E\ . + T$
I_1

$E \to T\ .$
$T \to T\ . * F$
I_2

$T \to F\ .$
I_3

$F \to (\ .\ E)$
$E \to .E + T$
$E \to .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to \ .id$
I_4

$F \to id\ .$
I_5

$E \to E + .\ T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to \ .id$
I_6

$T \to T * .\ F$
$F \to .(E)$
$F \to \ .id$
I_7

$F \to (E\ .)$
$E \to E\ . + T$
I_8

$E \to E + T\ .$
$T \to T\ . * F$
I_9

$T \to T * F\ .$
I_{10}

$F \to (E)\ .$
I_{11}

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 [$A \to \alpha.a\beta$] 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 $[A \to \alpha.a\beta]$ 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 $[A \to \alpha.a\beta]$ 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 [$A \to \alpha.a\beta$] 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 [$A \to \alpha.a\beta$] 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 [$A \to \alpha.a\beta$] 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 [$A \to \alpha.a\beta$] 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 $[A \rightarrow \alpha.]$ is in I_i

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

For all terminals in Follow(E→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 [$A \to \alpha.$] is in I_i

Action[i , a] = “reduce $A \to \alpha$” 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 [$A \to \alpha.$] is in I_i

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

For all terminals in Follow(T→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 [$A \to \alpha.$] is in I_i

Action[i , a] = “reduce $A \to \alpha$” for all a ∈ 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			

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 $A \rightarrow B$
            i.   Pop $|B|$ symbols off the stack
            ii.  Push GOTO[$t$, $A$] on to the stack
            iii. Output production $A \rightarrow B$
      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**

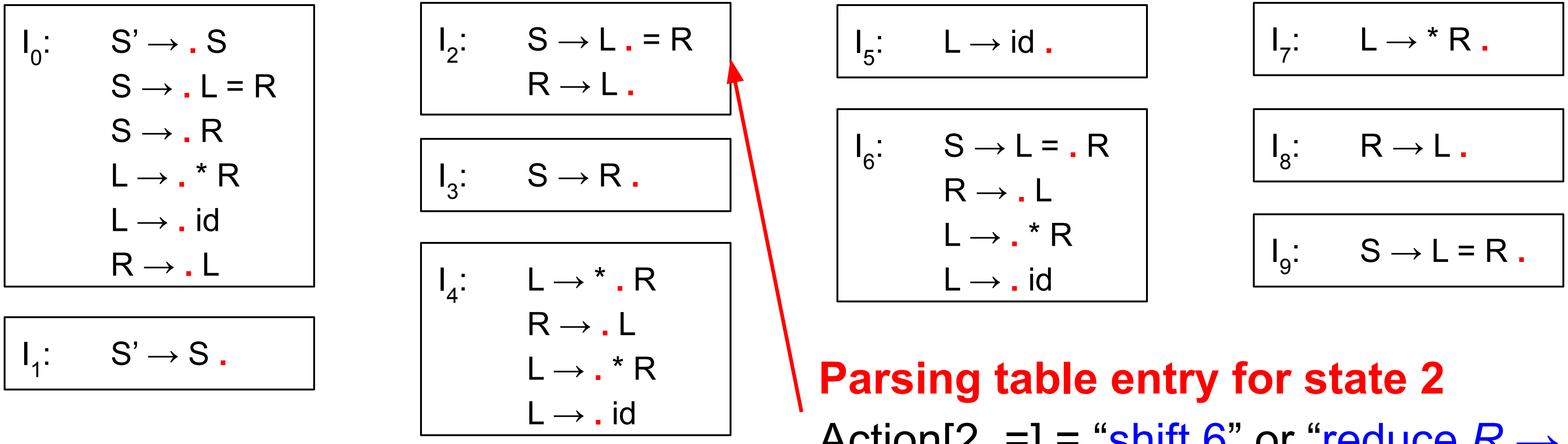
LR(0) Automation: Another example

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

$L \to {*}R \mid id$

$R \to L$

Construct SLR parser for the above grammar



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

I_1: $S' \to S\ .$

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

I_3: $S \to R\ .$

I_4: $L \to {*}\ .\ R$
$R \to .\ L$
$L \to .\ {*}\ R$
$L \to .\ id$

I_5: $L \to id\ .$

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

I_7: $L \to {*}\ R\ .$

I_8: $R \to L\ .$

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

Parsing table entry for state 2

Action[2, =] = “shift 6” or “reduce *R* → *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 → L = R | R
L → *R | id
R → L

Not SLR(1)

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

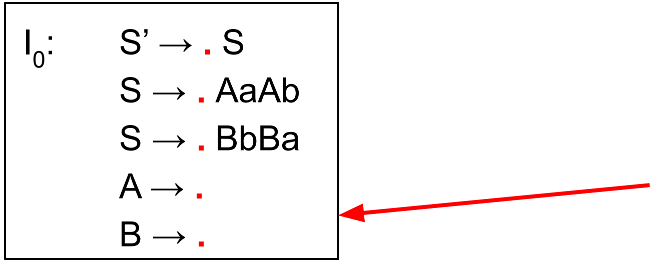
LR(0) Automation: Another example - 2

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

$A \to \varepsilon$

$B \to \varepsilon$

Construct SLR parser for the above grammar



I_0: $S' \to .\ S$

$S \to .\ AaAb$

$S \to .\ BbBa$

$A \to .$

$B \to .$

Parsing table entry for state I_0

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

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