Bottom up Parsing

Bottom up Parsers

Top-down parsers (LL(1), recursive descent)

- Start at the root of the parse tree from the start symbol and grow toward leaves (similar to a derivation)
- Pick a production and try to match the input
- Bad “pick” $\Rightarrow$ may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*
- **Bottom-up parsers (LR(1), operator precedence**)
- Start at the leaves and grow toward root
- We can think of the process as reducing the input string to the start symbol
- At each reduction step a particular substring matching the right-side of a production is replaced by the symbol on the left-side of the production
- Bottom-up parsers handle a large class of grammars

Bottom up Parsing

- A parse tree is created from the leaf upwards
- The symbols in the input are placed at the leaf nodes of the tree
- Starting from the leaves, the parser fills in the internal nodes of the parse tree gradually eventually finding the root.
- The creation of an internal node involves replacing symbols from (VUT)* by a single non terminal repeatedly
- The task is to identify the RHS of a production rule in the tree constructed so far and replace it with the corresponding LHS of the production rule
- This is called **reduction**
    - The crucial task in bottom up parsing is to find productions that have to be used for reduction

Shift Reduce Parsing

- A generic name for a family pf parsers that employ the strategy of bottom up parsing.

- Bottom-up parsing is also known as **shift-reduce parsing** because its two main actions are shift and reduce
  - At each **shift action**, the current symbol in the input string is pushed to a stack
  - At each **reduction step**, the symbols at the top of the stack (this symbol sequence is the right side of a production) will be replaced by the non-terminal at the left side of that production.
  - There are also two more actions: **accept and error**.

Shift-Reduce Parsing

- A shift-reduce parser tries to reduce the given input string into the starting symbol.

  a string $\xrightarrow[\textit{reduced to}]{}$ the starting symbol

- At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule.

- If the substring is chosen correctly, the right most derivation of that string is created in the reverse order.

Rightmost Derivation:	$S \Rightarrow \omega$
Shift-Reduce Parser finds:	$\omega \Leftarrow \ldots \Leftarrow S$

Handles

- **Handle of a string**: Substring that matches the RHS of some production AND whose reduction to the non-terminal on the LHS represents one step along the reverse of a rightmost derivation.

- A **handle** of a right sentential form γ ($\equiv \alpha\beta\omega$) is a production rule $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ.

$$S \Rightarrow \alpha A\omega \Rightarrow \alpha\beta\omega$$

i.e. $A \rightarrow \beta$ is a handle of $\alpha\beta\gamma$ at the location immediately after the end of α,

- If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.
- The string ω to the right of the handle contains terminal symbols only

Example

Consider:

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

$$S \Rightarrow \underline{aABe} \Rightarrow aA\underline{d}e \Rightarrow a\underline{Abc}de \Rightarrow a\underline{b}bcde$$

$S \rightarrow aABe$ is a handle of $\underline{aABe}$ in location 1.
$B \rightarrow d$ is a handle of $aA\underline{d}e$ in location 3.
$A \rightarrow Abc$ is a handle of $a\underline{Abc}de$ in location 2.
$A \rightarrow b$ is a handle of $a\underline{b}bcde$ in location 2.

Handle

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

- **Rightmost Derivation**

- $E \underset{rm}{\Rightarrow} E+\underline{E}$

  $\underset{rm}{\Rightarrow} E+E*\underline{E}$

  $\underset{rm}{\Rightarrow} E+\underline{E}*id3$

  $\underset{rm}{\Rightarrow} \underline{E}+id2*id3$

  $\underset{rm}{\Rightarrow} id1+id2*id3$

- **Shift Reduce Parsing**

$\underline{id1}+id2*id3$

$E+\underline{id2}*id3$

$E+E*\underline{id3}$

$E+\underline{E*E}$

$\underline{E+E}$

E

Handle Pruning

- A rightmost derivation in reverse can be obtained by handle pruning
- Start with a string of terminals, ω that we wish to parse
- If ω is a sentence of the grammar, then we start with γ_n, where γ_n is the nth right sentential form of some at yet unknown rightmost derivation

- $S=\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = \omega$

Apply the following simple algorithm

- Start from γ_{n} , locate β_n in γ_n and replace with the LHS of the production $A_n \rightarrow \beta_n$ to obtain the previous right sentential form γ_{n-1}.
- Then find a β_{n-1} in γ_{n-1} and reduce this handle with the LHS of the production $A_{n-1} \rightarrow \beta_{n-1}$ to get γ_{n-2}.
- Repeat this, until we reach S.

Stack Implementation of Shift Reduce Parser

- Shift reduce parser can be implemented with a stack and an input buffer
- **Initial configuration:** Stack: \$, Input: w\$
  - Parser operates by shifting zero or more input symbols onto the stack until a handle β is on top of the stack.
  - Parser then reduces β to the left side of the appropriate production
  - Parser repeats this process until it has discovered an error or until the stack contains the start symbol and the input is empty.
- **Final configuration:** Stack: \$S, Input: \$
  - The parser halts and announces successful completion of parsing

Stack Implementation of Shift Reduce Parsing

Four possible actions of a shift reduce parser

1. **Shift** : The next input symbol is shifted onto the top of the stack.

2. **Reduce**: The right end of the string to be reduced is on top of the stack.
   Locate the left end of the string within the stack. Replace the handle on the
   top of the stack by the nonterminal.

3. **Accept**: Successful completion of parsing.

4. **Error**: Parser discovers a syntax error, and calls an error recovery routine.

Stack Implementation of Shift Reduce Parsing

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	Reduce F→id
$F	+id*id$	Reduce T→F
$T	+id*id$	Reduce E→T
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	Reduce F→id
$E+F	*id$	Reduce T→F
$E+T	*id$	shift
$E+T*	id$	shift
$E+T*id	$	Reduce F→id
$E+T*F	$	Reduce T→T*F
$E+T	$	Reduce E→E+T
$E	$	Accept

Conflicts During Shift Reduce Parsing

- Stack contents and the next input symbol may not decide action
    - **shift/reduce conflict**: Whether to make a shift operation or a reduction.
    - **reduce/reduce conflict**: The parser cannot decide which of several reductions to make.

- There are context-free grammars for which shift-reduce parsers cannot be used.
- If a shift-reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar.
    - L: left to right scanning
    - R: rightmost derivation in reverse
    - k: lookahead

- An ambiguous grammar can never be a LR grammar.

Shift Reduce Parsers

- Two main categories of shift reduce parsers

1. Operator Precedence Parser
    - simple, but only a small class of grammars.
2. LR Parser
    - covers wide range of grammars.
        - SLR – simple LR parser
        - Canonical LR – most general LR parser
        - LALR – intermediate LR parser (lookahead LR parser)
    - SLR, Canonical LR and LALR work same, only their parsing tables are different.

Operator Precedence Parser

- **Operator grammar**
  - small, but an important class of grammars
- In an *operator grammar*, no production rule can have:
  - ε at the right side
  - two adjacent non-terminals at the right side.

- Ex:

$E \rightarrow AB$	$E \rightarrow EAE$	$E \rightarrow E+E$
$A \rightarrow a$	$E \rightarrow id$	$\mid E*E$
$B \rightarrow b$	$A \rightarrow + \mid * \mid /$	$\mid E/E \mid id$
not operator grammar	not operator grammar	operator grammar

Precedence Relations

- In operator-precedence parsing, we define three disjoint precedence relations between certain pairs of terminals.

a <· b	a yields precedence to b or b has higher precedence than a
a =· b	a has same precedence as b
a ·> b	a takes precedence over b or b has lower precedence than a

- The determination of correct precedence relations between terminals are based on the traditional notions of associativity and precedence of operators. (Unary minus causes a problem).
  - If * has higher precedence than +, then * ·>+ and + <· *

Using Operator-Precedence Relations

- The intention of the precedence relations is to find the handle of a right-sentential form,

  <· with marking the left end,

  =· appearing in the interior of the handle, and

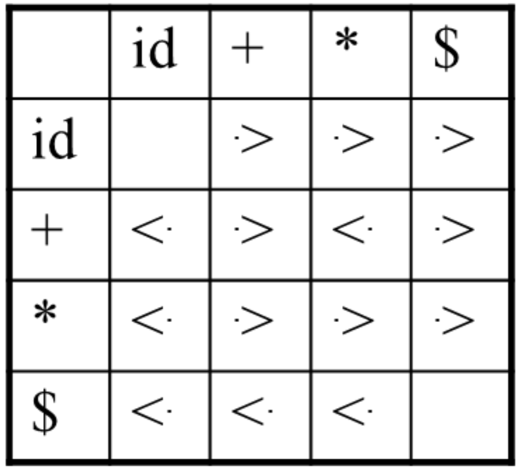
  ·> marking the right hand.

- Right sentential form: $\beta_0 a_1 \beta_1 a_2 \beta_2 \ldots\ldots a_n \beta_n$
  - Each β_i is a single nonterminal and each a_i is a single terminal
- In our input string, remove all nonterminals ($\beta_0, \beta_1, \beta_2 \ldots, \beta_n$) and place precedence relations between all pairs of terminals ($\$a_1 a_2 \ldots a_n\$$,)
  - $ <· b and b ·> $ for all terminals b

Using Operator-Precedence Relations

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

The partial operator-precedence table for this grammar



	id	+	*	$
id		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	

- Input string: id+id*id

- Insert precedence relation

$$\$ \lessdot id \gtrdot + \lessdot id \gtrdot * \lessdot id \gtrdot \$$$

To Find The Handles

1. Scan the string from left end until the first ·> is encountered.
2. Then scan backwards (to the left) over any =· until a <· is encountered.
3. The handle contains everything to left of the first ·> and to the right of the <· encountered in step 2 including any intervening or surrounding nonterminal

$ <· id ·> + <· id ·> * <· id ·> $	E → id	$ id + id * id $
$ <· + <· id ·> * <· id ·> $	E → id	$ E + id * id $
$ <· + <· * <· id ·> $	E → id	$ E + E * id $
$ <· + <· * ·> $	E → E*E	E + E * ·E $
$ <· + ·> $	E → E+E	E + E $
$ $		$ E $

Operator Precedence Parsing Algorithm

Input: An input string w$ and a table of precedence relations

Output: If *w* is well formed, a skeletal parse tree with placeholder nonterminal E labelling all interior nodes, otherwise an error indication

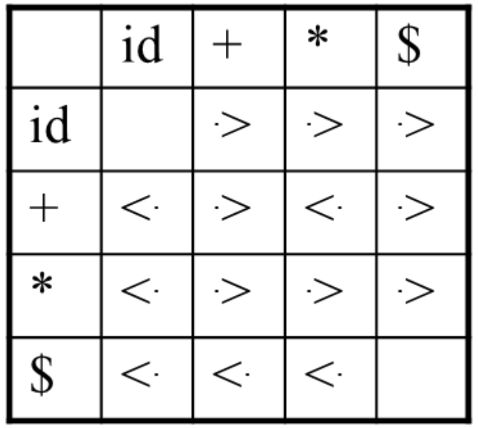
Method: Initially stack contains $ and input buffer the string w$

Algorithm:

```
set ip to point to the first symbol of w$ ;
repeat forever
   if ( $ is on top of the stack and ip points to $ ) then return
   else begin
      let a be the symbol on top of the stack and let b be the symbol pointed to by ip;
      if ( a <· b  or  a =· b ) then begin        /* SHIFT */
         push b onto the stack;
         advance ip to the next input symbol;
     end
     else if ( a ·> b ) then                      /* REDUCE */
         repeat pop stack
         until the top of stack terminal is related by <· to the terminal most recently popped ;
      else error();
   end
```

Operator Precedence Parsing -Example

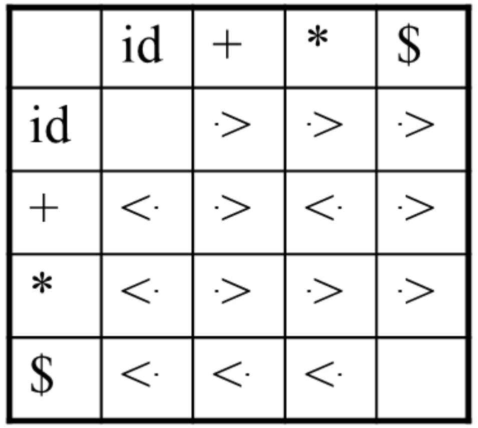
stack	***input***	***action***
$	id+id$	$ <· id shift
$id	+id$	id ·> +, reduce E → id
$	+id$	$ <· +, shift
$+	id$	+ <· id, shift
$+id	$	id ·> $, reduce E → id
$+	$	$ <· +, + ·> $, reduce E → E+E
$	$	accept



	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Operator Precedence Parsing -Example

stack	***input***	***action***
\$	id+id*id\$	\$ $\lessdot$ id shift
\$id	+id*id\$	id $\gtrdot$ + reduce E $\longrightarrow$ id
\$	+id*id\$	\$ $\lessdot$ + shift
\$+	id*id\$	+ $\lessdot$ id shift
\$+id	*id\$	id $\gtrdot$ * reduce E $\longrightarrow$ id
\$+	*id\$	+ $\lessdot$ * shift
\$+*	id\$	* $\lessdot$ id shift
\$+*id	\$	id $\gtrdot$ \$ reduce E $\longrightarrow$ id
\$+*	\$	* $\gtrdot$ \$, pop *, reduce E $\longrightarrow$ E*E
\$+	\$	+ $\gtrdot$ \$, \$ $\lessdot$ +, reduce E $\longrightarrow$ E+E
\$	\$	accept



	id	+	*	\$
id		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
\$	$\lessdot$	$\lessdot$	$\lessdot$	

How to Create Operator-Precedence Relations

We use associativity and precedence relations among operators.

1. If operator θ_1 has higher precedence than operator θ_2, ➔ $\theta_1 \cdot> \theta_2$ and $\theta_2 <\cdot \theta_1$

2. If operator θ_1 and operator θ_2 have equal precedence,
   they are left-associative ➔ $\theta_1 \cdot> \theta_2$ and $\theta_2 \cdot> \theta_1$
   they are right-associative ➔ $\theta_1 <\cdot \theta_2$ and $\theta_2 <\cdot \theta_1$

3. For all operators θ,
   $\theta <\cdot \text{id}$, $\text{id} \cdot> \theta$, $\theta <\cdot ($, $(<\cdot \theta$, $\theta \cdot>)$, $) \cdot> \theta$, $\theta \cdot> \$$, and $\$ <\cdot \theta$

4.	$(\doteq)$	$\$ <\cdot ($	$\text{id} \cdot>)$	$) \cdot>$	
	$(<\cdot ($	$\$ <\cdot \text{id}$	$\text{id} \cdot> \$$	$) \cdot>)$	$(<\cdot \text{id}$

LR Parser

LR Parser

- Efficient bottom up parser for a large class of grammar

LR(k)

left to right scanning

right-most derivation in reverse

k lookahead (k is omitted ➔ it is 1)

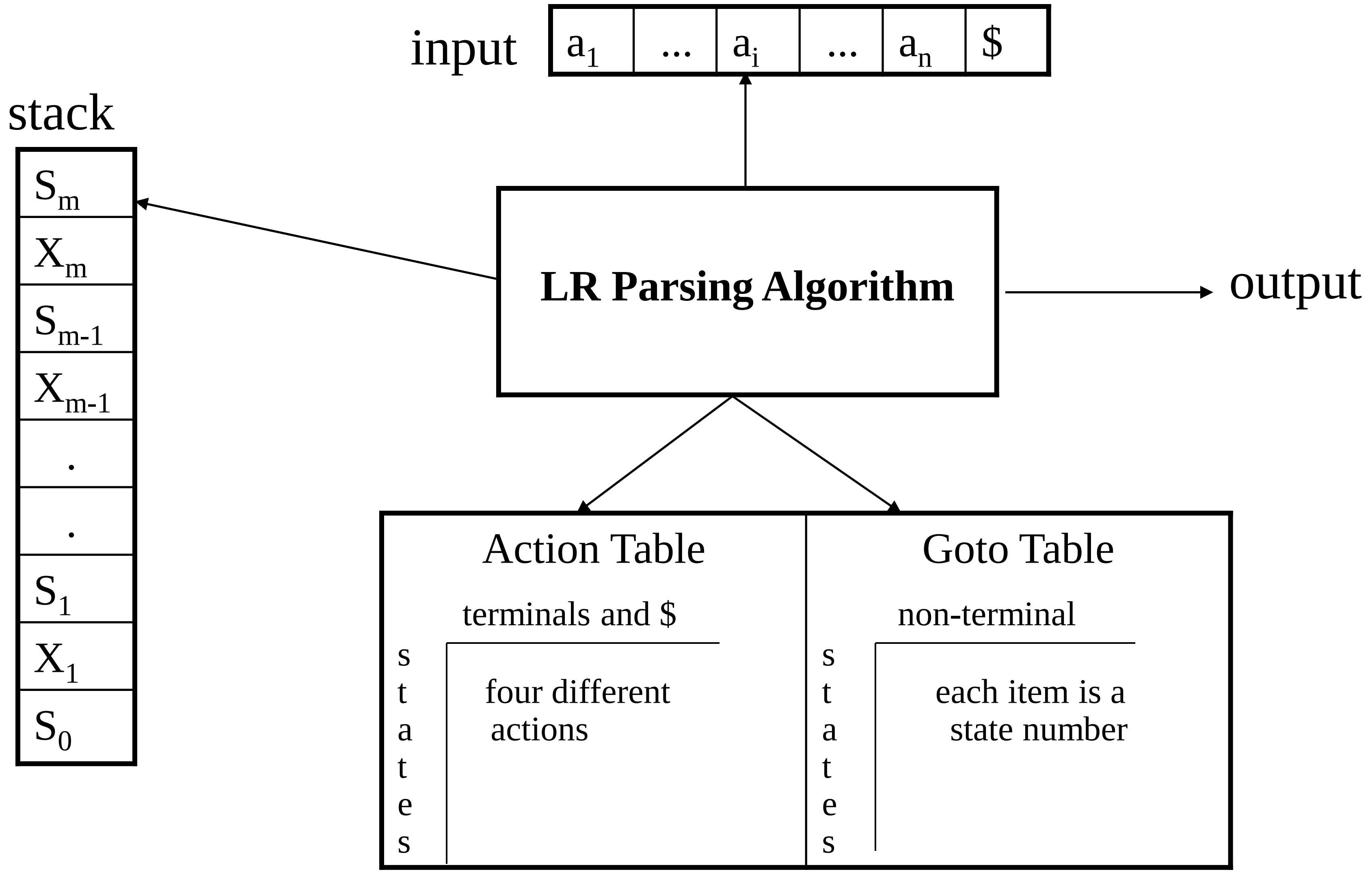
- LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing
  - The class of grammars that can be parsed using LR methods is a **proper superset** of the class of grammars that can be parsed with predictive parsers.

    LL(1)-Grammars $\subset$ LR(1)-Grammars
  - An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.
  - Can recognize virtually all programming language constructs for which CFG can be written
- Drawback: Too much work to construct LR parser by hand
  - Specialized parser generators are required

LR Parser

- LR Parsers
  - covers wide range of grammars.
  - SLR – simple LR parser
  - Canonical LR – most general LR parser
  - LALR – look-ahead LR parser (intermediate LR parser)
  - SLR, Canonical LR and LALR work same (they used the same algorithm), only their parsing tables are different.
- LR Parser consists of two parts: driver routine and parsing table
  - Driver routine same for all LR parser
  - Parsing table is different

LR Parser


input
a_1
...
a_i
...
a_n
$
stack
S_m
X_m
S_{m-1}
X_{m-1}
.
.
S_1
X_1
S_0
LR Parsing Algorithm
output
Action Table
terminals and $
states
four different actions
Goto Table
non-terminal
states
each item is a state number

SLR Parsing Table for Grammar, G

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

- sj: shift current input symbol and state j onto stack
- rj: reduce by production number j
- acc: accept
- Blank entries: error

	Action						Goto		
state	**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			

A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:

$$(\underbrace{S_o X_1 S_1 \ldots X_m S_m}_{\text{Stack}},\ \underbrace{a_i a_{i+1} \ldots a_n \$}_{\text{Rest of Input}})$$

- A configuration of a LR parsing represents the right sentential form:

$$X_1 \ldots X_m a_i a_{i+1} \ldots a_n \$$$

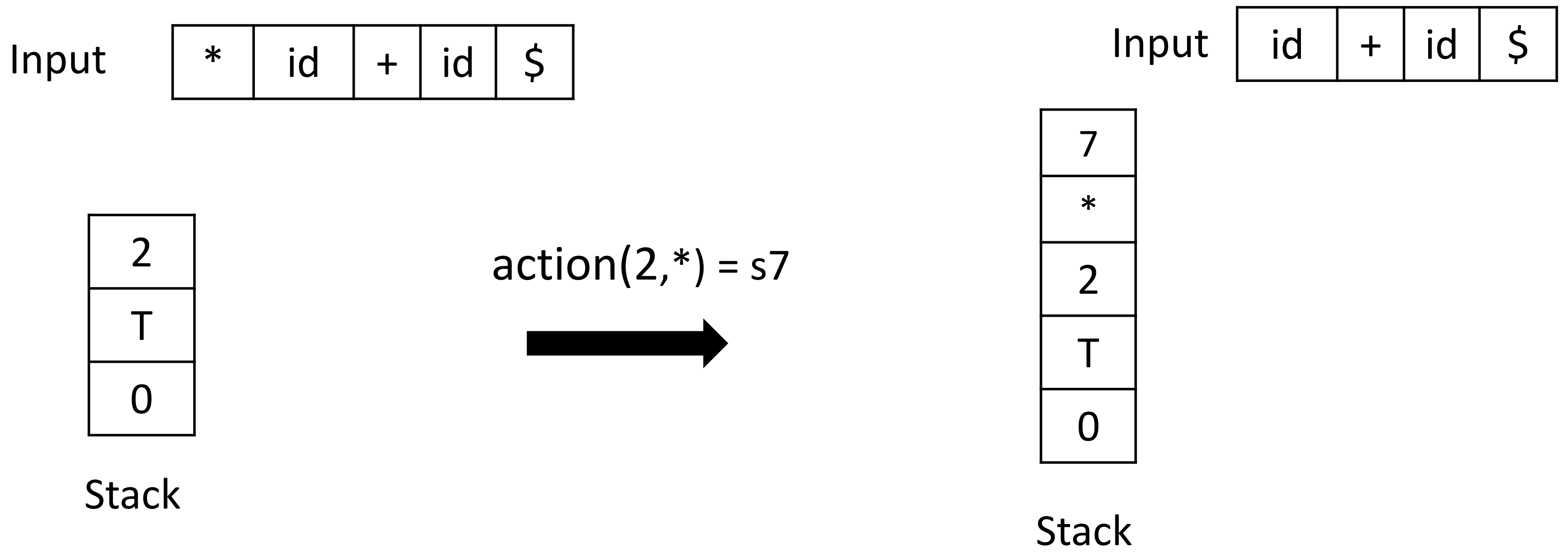
- Initial Stack contains just S_o
- S_m and a_i decides the parser action by consulting the parsing action table entry *action[S_m,a_i]*

LR Parser

1. **action[S_m, a_i] = shift S**

   shifts the next input symbol and the state **S** onto the stack

   (S_o X_1 S_1 … X_m S_m, a_i a_{i+1} … a_n \$) $\rightarrow$ (S_o X_1 S_1 … X_m S_m a_i S, a_{i+1} … a_n \$)



LR Parser

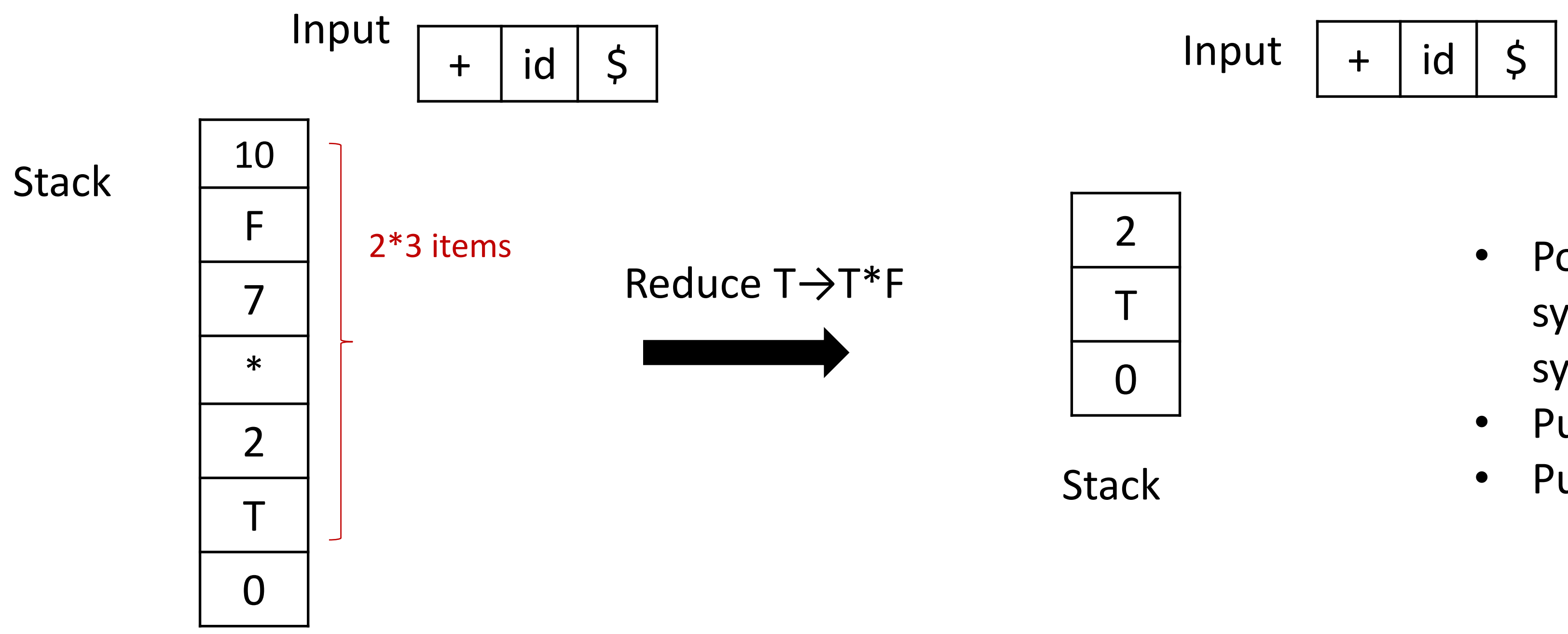
2. **action[S_m, a_i] = reduce A→β**

    pop 2|β| (=2r) items from the stack;

    then push **A** and **S** where **S = goto[s_{m-r} , A]**

    (S_o X_1 S_1 … X_m S_m, a_i a_{i+1} … a_n $) → (S_o X_1 S_1 … X_{m-r} S_{m-r} A S, a_i … a_n $)

- Output the reducing production reduce A→β



- Pop 3 grammar symbols and 3 state symbols
- Push T
- Push goto(0,T)

LR Parser

3. **If action[S_m , a_i] = accept** , Parsing successfully completed

4. **If action[S_m , a_i]** = **error**, Parser detected an error (an empty entry in the action table) and calls an error recovery routine.

LR Parsing Algorithm

- **Input:** An input string w and an LR parsing table with action and goto for grammar G
- **Output:** If w is in L(G), a bottom up parse for w, otherwise an error indication
- **Method:** Initially the parser has S_0 on the stack where S_0 is the initial state and w$ in the input buffer

```
set ip to point to the first symbol in w$
repeat forever begin
let 's' be state on top of the stack  and  'a' be symbol pointed to by ip
if action[s,a] = shift s' then begin
    push a then s' onto stack
    advance ip to next input symbol
end
else if action[s,a] = reduce $A \rightarrow \beta$ then begin
    pop 2*| $\beta$ | symbols off stack
    let s' be state now on top of stack
    push A, then push goto[s',A] onto stack
    output production $A \rightarrow \beta$
end
else if action[s,a] = accept
    return success
else
    error()
end
```

Moves of LR Parser on id*id+id			
Stack	**Input**	**Action**	**Output**
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

1) $E \to E+T$
2) $E \to T$
3) $T \to T*F$
4) $T \to F$
5) $F \to (E)$
6) $F \to id$

state	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 Grammar

- A grammar for which we can construct an LR parsing table in which every entry is uniquely defined is an LR grammar.
- For an LR grammar, the shift reduce parser should be able to recognize handles when they appear on top of the stack.
- Each state symbol summarizes the information contained in the stack below it.
- The LR parser can determine from the state on top of the stack everything it needs to know about what is in the stack.
- The next k input symbols can also help the LR parser to make shift reduce decisions.
- Grammar that can be parsed by an LR parser by examining upto k input symbols on each move is called an LR(k) grammar.

Constructing SLR Parsing Tables – LR(0) Item

- LR parser using SLR parsing table is called an SLR parser.
- A grammar for which an SLR parser can be constructed is an SLR grammar.
- **<u>LR(0) item</u>:**
- An LR(0) item (item) of a grammar G is a production of G with a dot at the some position on the right side.
- Eg: $A \rightarrow XYZ$ Possible LR(0) Items: (four different possibility)

  $A \rightarrow .XYZ$
  $A \rightarrow X.YZ$
  $A \rightarrow XY.Z$
  $A \rightarrow XYZ.$

- Sets of LR(0) items will be the states of action and goto table of the SLR parser.
- The production $A \rightarrow \varepsilon$ yields only one item $A \rightarrow .$

Constructing SLR Parsing Tables – LR(0) Item

- **Canonical LR(0) collection**:
  - A collection of sets of LR(0) items
  - Basis for constructing SLR parsers.
- To construct the canonical LR(0) collection for a grammar we define an **augmented grammar** and **two functions- closure and goto**.
- **Augmented Grammar:**
  - If G is grammar with start symbol S, then augmented grammar G' is grammar G with a new production rule $S' \rightarrow S$ where S' is the new starting symbol. i.e   G U {S' → S}
  - The start state of G' = S'
  - This is done to signal to the parser when the parsing should stop parsing and announce acceptance of input.

Constructing SLR Parsing Tables – LR(0) Item

- **Complete and Incomplete Items**:
    - An LR(0) item is complete if ‘.’ is the last symbol in RHS else it is incomplete.
    - For every rule $A \rightarrow \alpha$, $\alpha \neq \varepsilon$, there is only one complete item $A \rightarrow \alpha.$, but as many incomplete items as there are grammar symbols in the RHS.

Kernel and Non-Kernel items:

- An LR(0) item is a kernel item if the dot is not at the left end.
- $S' \rightarrow .S$ is an exception and is considered to be a kernel item.
- Non-kernel items are the items which have the dot at leftmost end.
- Sets of items are formed by taking the closure of a set of kernel items.

The Closure Operation

- If ***I*** is a set of LR(0) items for a grammar G, then ***closure(I)*** is the set of LR(0) items constructed from I by the two rules:

1. Initially, every LR(0) item in I is added to closure(I).
2. If $\mathbf{A \to \alpha.B\beta}$ is in closure(I) and $\mathbf{B \to \gamma}$ is a production rule of G; then add $\mathbf{B \to .\gamma}$ to closure(I) if it is not already there.

We will apply this rule until no more new LR(0) items can be added to closure(I).

The Closure Operation

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

$$\text{closure}(\{E' \rightarrow \bullet E\}) = \{\, E' \rightarrow \bullet E \quad \{\text{rule 1}\}$$

$$\left.\begin{array}{l} E \rightarrow \bullet E+T \\ E \rightarrow \bullet T \\ T \rightarrow \bullet T*F \\ T \rightarrow \bullet F \\ F \rightarrow \bullet (E) \\ F \rightarrow \bullet id \ \} \end{array}\right\} \{\text{rule 2}\}$$

Computation of Closure

```
function closure ( I )
begin
        J := I;
        repeat
                for each item $\mathbf{A \rightarrow \alpha.B\beta}$ in J and each production
                $\mathbf{B \rightarrow}\gamma$ of G such that $\mathbf{B \rightarrow .}\gamma$ is not in J do
                        add $\mathbf{B \rightarrow .}\gamma$ to J
        until no more items can be added to J
       return J
End
```

Goto Operation

- If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha.X\beta$ in I then every item in **closure([$A \rightarrow \alpha X.\beta$])** will be in goto(I,X).
  - If I is the set of items that are valid for some viable prefix γ, then goto(I,X) is the set of items that are valid for the viable prefix γX.

I ={ $E' \rightarrow E.$, $E \rightarrow E.+T$}

goto(I,+) = { $E \rightarrow E+.T$

$$\left.\begin{array}{l} T \rightarrow .T*F \\ T \rightarrow .F \\ F \rightarrow .(E) \\ F \rightarrow .id \quad \} \end{array}\right\} \quad \text{closure}(E \rightarrow E+.T)$$

Example

$I = \{E' \to .E$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id \}$

goto (I, E) = $\{ E' \to E.$
$E \to E.+T \}$

goto (I, T) = $\{ E \to T.$
$T \to T.*F \}$

goto (I, F) = $\{T \to F. \}$

goto(I, id) = $\{ F \to id. \}$

goto (I, ( ) = $\{ F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id \}$

Sets of Items Construction

Algorithm to construct C, the canonical collection of LR(0) items for the augmented grammar G’

```
Procedure items( G’ )
begin
    C:= { closure({[S’→.S]}) }
    **repeat**
    **for each** set of items I in **C** and each grammar symbol X
            **if** goto(I,X) is not empty and not in ***C***
                add goto(I,X) to ***C***
    until no more sets of LR(0) items can be added to ***C***.
End
```

- goto function is a DFA on the sets in C.

Canonical Collection of LR(0) Items

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

I_1: goto(I_0, E)
$E' \rightarrow E.$
$E \rightarrow E.+T$

I_2: goto(I_0,T)
$E \rightarrow T.$
$T \rightarrow T.*F$

I_3: goto(I_0, F)
$T \rightarrow F.$

I_4: goto(I_0, ()
$F \rightarrow (.E)$
$E \rightarrow .E+T$
$E \rightarrow .T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_5: goto(I_0, id)
$F \rightarrow id.$

I_6: goto(I_1, +)
$E \rightarrow E+.T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_7: goto(I_2, *)
$T \rightarrow T*.F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_8: goto(I_4, E)
$F \rightarrow (E.)$
$E \rightarrow E.+T$

goto(I_4, T)=I_2
goto(I_4, F)=I_3
goto(I_4, ()=I_4
goto(I_4, id)=I_5

I_9: goto(I_6, T)
$E \rightarrow E+T.$
$T \rightarrow T.*F$

goto(I_6, F)=I_3
goto(I_6,()=I_4
goto(I_6, id)=I_5

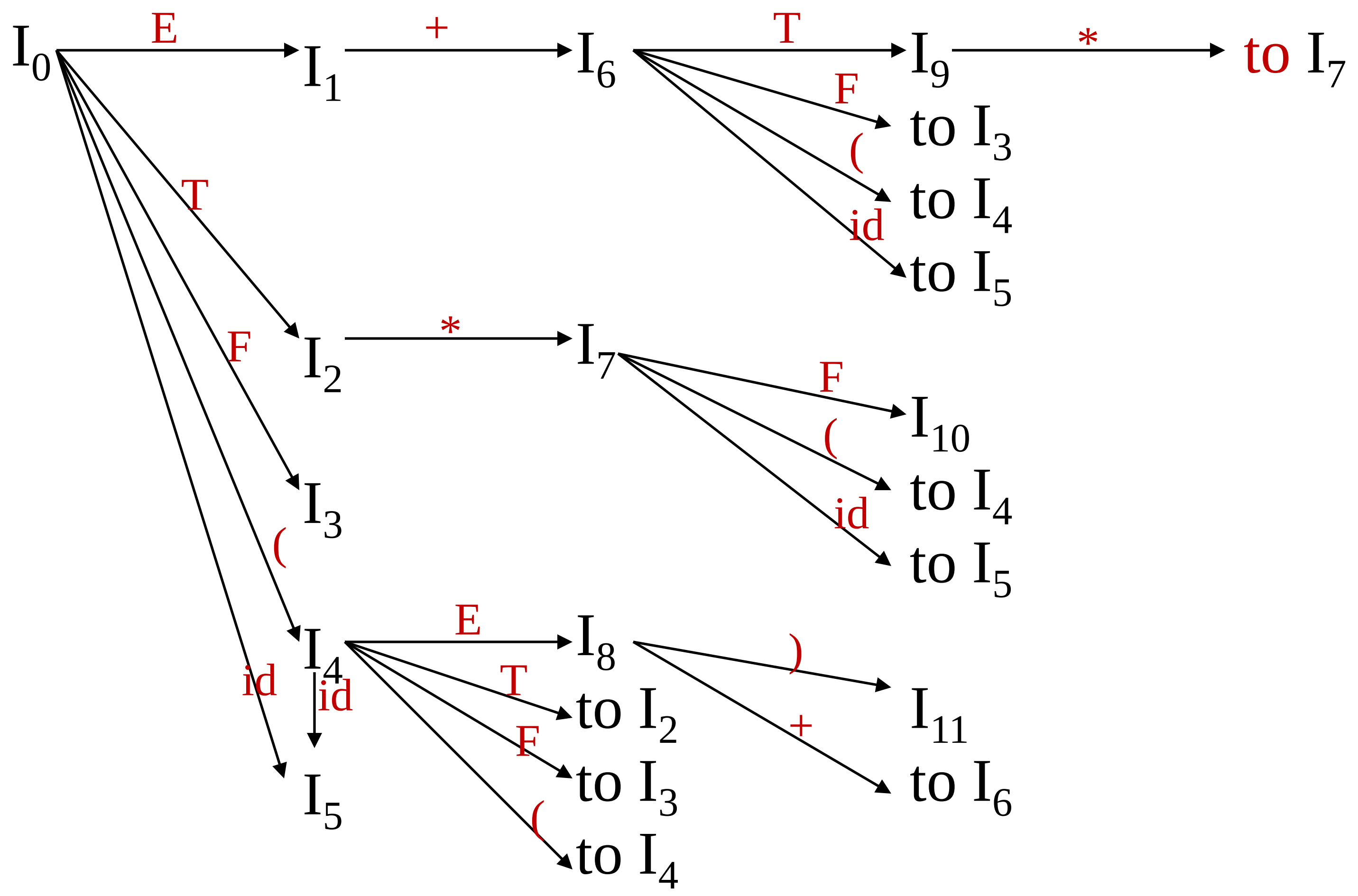
I_{10} :goto(I_7, F)
$T \rightarrow T*F.$

goto(I_7,()=I_4
goto(I_7, id)=I_5

I_{11} :goto(I_8,))
$F \rightarrow (E).$

goto(I_8,+) = I_6
goto(I_9,*) = I_7

Transition Diagram (DFA) of Goto Function



Constructing SLR Parsing Table

Input: An augmented grammar G'

Output: The SLR parsing functions action and goto for grammar G'

1. Construct the canonical collection of sets of LR(0) items for G', **C={I_0,...,I_n}**
2. State i is constructed from I_i . The **parsing actions** for state i are determined as follows:
   - If **[$A \rightarrow \alpha.a\beta$] is in I_i** and goto(I_i,a)=I_j , then action[i,a] is ***shift j*.** Here a should be a terminal
   - If **[$A \rightarrow \alpha.$]** is in I_i , then set **action[i,a]** to ***reduce $A \rightarrow \alpha$*** for all a in **FOLLOW(A)** where **$A \neq S'$**.
   - If **[$S' \rightarrow S.$]** is in I_i , then action[i,$] is ***accept***.

   If any conflicting actions generated by these rules, the grammar is not SLR(1).
3. The **goto transitions** for state I are constructed for all nonterminals A using the rule

   If goto(I_i,A)=I_j then goto[i,A]=j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser is the one constructed from the sets of items containing [$S' \rightarrow .S$]

Canonical Collection of LR(0) Items

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

I_1: goto(I_0, E)
$E' \rightarrow E.$
$E \rightarrow E.+T$

I_2: goto(I_0,T)
$E \rightarrow T.$
$T \rightarrow T.*F$

I_3: goto(I_0, F)
$T \rightarrow F.$

I_4: goto(I_0, ()
$F \rightarrow (.E)$
$E \rightarrow .E+T$
$E \rightarrow .T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_5: goto(I_0, id)
$F \rightarrow id.$

I_6: goto(I_1, +)
$E \rightarrow E+.T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_7: goto(I_2, *)
$T \rightarrow T*.F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_8: goto(I_4, E)
$F \rightarrow (E.)$
$E \rightarrow E.+T$

goto(I_4, T)=I_2
goto(I_4, F)=I_3
goto(I_4, ()=I_4
goto(I_4, id)=I_5

I_9: goto(I_6, T)
$E \rightarrow E+T.$
$T \rightarrow T.*F$

goto(I_6, F)=I_3
goto(I_6,()=I_4
goto(I_6, id)=I_5

I_{10} :goto(I_7, F)
$T \rightarrow T*F.$

goto(I_7,()=I_4
goto(I_7, id)=I_5

I_{11} :goto(I_8,))
$F \rightarrow (E).$

goto(I_8,+) = I_6
goto(I_9,*) = I_7

SLR Parsing Table

Action Goto

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

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

FOLLOW(E)= {+,),$}
FOLLOW(T)= {*,+,),$}
FOLLOW(F)= {*,+,),$}

goto(I_0, E)= I_1
goto(I_0,T)= I_2
goto(I_0, F)= I_3
goto(I_0, ()= I_4
goto(I_0, id)= I_5
goto(I_1, +)= I_6
goto(I_2, *)= I_7
goto(I_4, E)= I_8
goto(I_4, T)=I_2
goto(I_4, F)=I_3
goto(I_4, ()=I_4
goto(I_4, id)=I_5

goto(I_6, T)= I_9
goto(I_6, F)=I_3
goto(I_6,()=I_4
goto(I_6, id)=I_5
goto(I_7, F)= I_{10}
goto(I_7,()=I_4
goto(I_7, id)=I_5
goto(I_8,))= I_{11}
goto(I_8,+) = I_6
goto(I_9,*) = I_7

SLR(1) Grammar

- An LR parser using SLR(1) parsing tables for a grammar G is called as the SLR(1) parser for G.
- If a grammar G has an SLR(1) parsing table, it is called SLR(1) grammar (or SLR grammar in short).
- Every SLR grammar is unambiguous, but every unambiguous grammar is not a SLR grammar.

shift/reduce and reduce/reduce conflicts

- If a state does not know whether it will make a shift operation or reduction for a terminal, we say that there is a **shift/reduce conflict**.

- If a state does not know whether it will make a reduction operation using the production rule `i` or `j` for a terminal, we say that there is a **reduce/reduce conflict**.

- If the SLR parsing table of a grammar G has a conflict, we say that that grammar is not SLR grammar.

SLR Parser-Example

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

Conflict Example-1

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

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

I_1: goto (I_0, S)
$S' \rightarrow S.$

I_2: goto (I_0, L)
$S \rightarrow L.=R$
$R \rightarrow L.$

I_3: goto (I_0, R)
$S \rightarrow R.$

I_4: goto (I_0, *)
$L \rightarrow *.R$
$R \rightarrow .L$
$L \rightarrow .*R$
$L \rightarrow .id$

I_5: goto (I_0, id)
$L \rightarrow id.$

I_6: goto (I_2, =)
$S \rightarrow L=.R$
$R \rightarrow .L$
$L \rightarrow .*R$
$L \rightarrow .id$

goto(I_4, *) =I_4
goto(I_4, *) d=I_5

I_7: goto (I_4, R)
$L \rightarrow *R.$

I_8: goto (I_4, L)
$R \rightarrow L.$

I_9: goto (I_6, R)
$S \rightarrow L=R.$

goto(I_6, L) =I_8
goto(I_6, *) =I_4
goto(I_6, id) =I_5

Action[2,=] = shift 6
Action[2,=] = reduce by $R \rightarrow L$
[$S \Rightarrow L=R \Rightarrow *R=R$] so follow(R) contains, =

Conflict Example2

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

I_0: $S' \rightarrow .S$

$S \rightarrow .AaAb$

$S \rightarrow .BbBa$

$A \rightarrow .$

$B \rightarrow .$

I_0: $A \rightarrow .$

FOLLOW(A)={a,b}

action(0,a)=reduce by $A \rightarrow \varepsilon$

action(0,b)=reduce by $A \rightarrow \varepsilon$

I_0: $B \rightarrow .$

FOLLOW(B)={a,b}

action(0,a)=reduce by $B \rightarrow \varepsilon$

action(0,b)=reduce by $B \rightarrow \varepsilon$

Problem: reduce/reduce conflict

Canonical LR(1) Parser

- Most general technique for constructing an LR parsing table.
- In SLR method, the state i makes a reduction by $A \rightarrow \alpha$
  - if the set of items I_i contains the item $[A \rightarrow \alpha.]$ and
  - **a** is FOLLOW(A)
- In some situations, when state i appears on top of the stack and the viable prefix $\beta\alpha$ on the stack is such that
  - βA cannot be followed by the terminal **a** in a right-sentential form
- This means that making reduction in this case is not correct.

LR(1) Item

- To avoid some of invalid reductions, the states need to carry more information.
- By splitting states when necessary, we can arrange to have each state of an LR parser indicate exactly which input symbols can follow a *handle* α for which there is a possible reduction to A
- Extra information is put into a state by including a terminal symbol as a second component in an item.
- A LR(1) item is:

  $A \rightarrow \alpha.\beta,a$ where **a** is the look ahead of the LR(1) item (**a** is a terminal or the right end marker, $)

  - 1 refers to the length of the second component, the look ahead symbol

LR(1) Item

- The lookahead has no effect in an item of the form $[A \to \alpha.\beta,a]$, where β is not $\in$.
- An item of the form $[A \to \alpha.,a]$ calls for a reduction by $A \to \alpha$ only if the next input symbol is a.
- The set of such a's will be a subset of FOLLOW(A), but it could be a proper subset.
- A state will contain $A \to \alpha\bullet,a_1$ where $\{a_1,\ldots,a_n\} \subseteq$ FOLLOW(A)

  $$\begin{gathered} A \to \alpha\bullet,a_1 \\ \ldots \\ A \to \alpha\bullet,a_n \end{gathered}$$

- Can be written as $A \to \alpha\bullet, a_1/a_2/../a_n$

Construction of sets of LR(1) Items

```
function closure(I)
begin
repeat
    for each item [A→α.Bβ,a] in I
        each production B → γ in G',
        and each terminal b in FIRST(βa),
        such that [B → .γ, b] is not in I do
        add [B → .γ, b] to I
until no more items can be added to I
return I
end
```

Construction of sets of LR(1) Items

```
function goto(I,X)
begin
        let J be the sets of items ([$A \rightarrow \alpha X.\beta,a$])
        such that ([$A \rightarrow \alpha.X\beta,a$]) is in I
        return closure(J)
end
```

- If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha.X\beta,a$ in I then every item in **closure([$A \rightarrow \alpha X.\beta,a$])** will be in goto(I,X).

Construction of The Canonical LR(1) Collection

```
procedure items(G')
begin
C is { closure({[S'→.S,$]}) }
repeat
    for each set of items I in C and each grammar symbol X
        if goto(I,X) is not empty and not in C
            add goto(I,X) to C
  until no more set of LR(1) items can be added to C.
end
```

- goto function is a DFA on the sets in C.

Canonical LR(1) Collection- Example

1. $S' \rightarrow S$
2. $S \rightarrow CC$
3. $C \rightarrow cC$
4. $C \rightarrow d$

I_0:
$S' \rightarrow \bullet S, \$$
$S \rightarrow \bullet CC, \$$
$C \rightarrow \bullet cC, c/d$
$C \rightarrow \bullet d, c/d$

I_0: closure([$S' \rightarrow \bullet S, \$$])

$S' \rightarrow \bullet S, \$$
match with $A \rightarrow \alpha.B\beta,a$
$A=S', \alpha=\varepsilon, B=S, \beta= \varepsilon, a=\$$
FIRST(βa)=FIRST($\$$)={$\$$}
Hence, add the item

$S \rightarrow \bullet CC, \$$
to
closure([$S' \rightarrow \bullet S, \$$])

$S \rightarrow \bullet CC, \$$
match with $A \rightarrow \alpha.B\beta,a$
$A=S, \alpha=\varepsilon, B=C, \beta= C, a=\$$
FIRST(βa)=FIRST($C\$$)={c,d}

Hence, add the items
$C \rightarrow \bullet cC, c$
$C \rightarrow \bullet cC, d$
$C \rightarrow \bullet d, c$
$C \rightarrow \bullet d, d$

to
closure([$S' \rightarrow \bullet S, \$$])

Canonical LR(1) Collection- Example

$S' \to S$
$S \to CC$
$C \to cC$
$C \to d$

I_0: $S' \to \bullet S, \$$
$S \to \bullet CC, \$$
$C \to \bullet cC, c/d$
$C \to \bullet d, c/d$

I_1: goto(I_0, S)=
$S' \to S\bullet, \$$

I_2: goto(I_0, C) =
$S \to C\bullet C, \$$
$C \to \bullet cC, \$$
$C \to \bullet d, \$$

I_3: goto(I_0, c) =
$C \to c\bullet C, c/d$
$C \to \bullet cC, c/d$
$C \to \bullet d, c/d$

I_4: goto(I_0, d) =
$C \to d\bullet, c/d$

I_5: goto(I_2, C) =
$S \to CC\bullet, \$$

I_6: goto(I_2, c) =
$C \to c\bullet C, \$$
$C \to \bullet cC, \$$
$C \to \bullet d, \$$

I_7: goto(I_2, d) =
$C \to d\bullet, \$$

I_8: goto(I_3, C) =
$C \to cC\bullet, c/d$

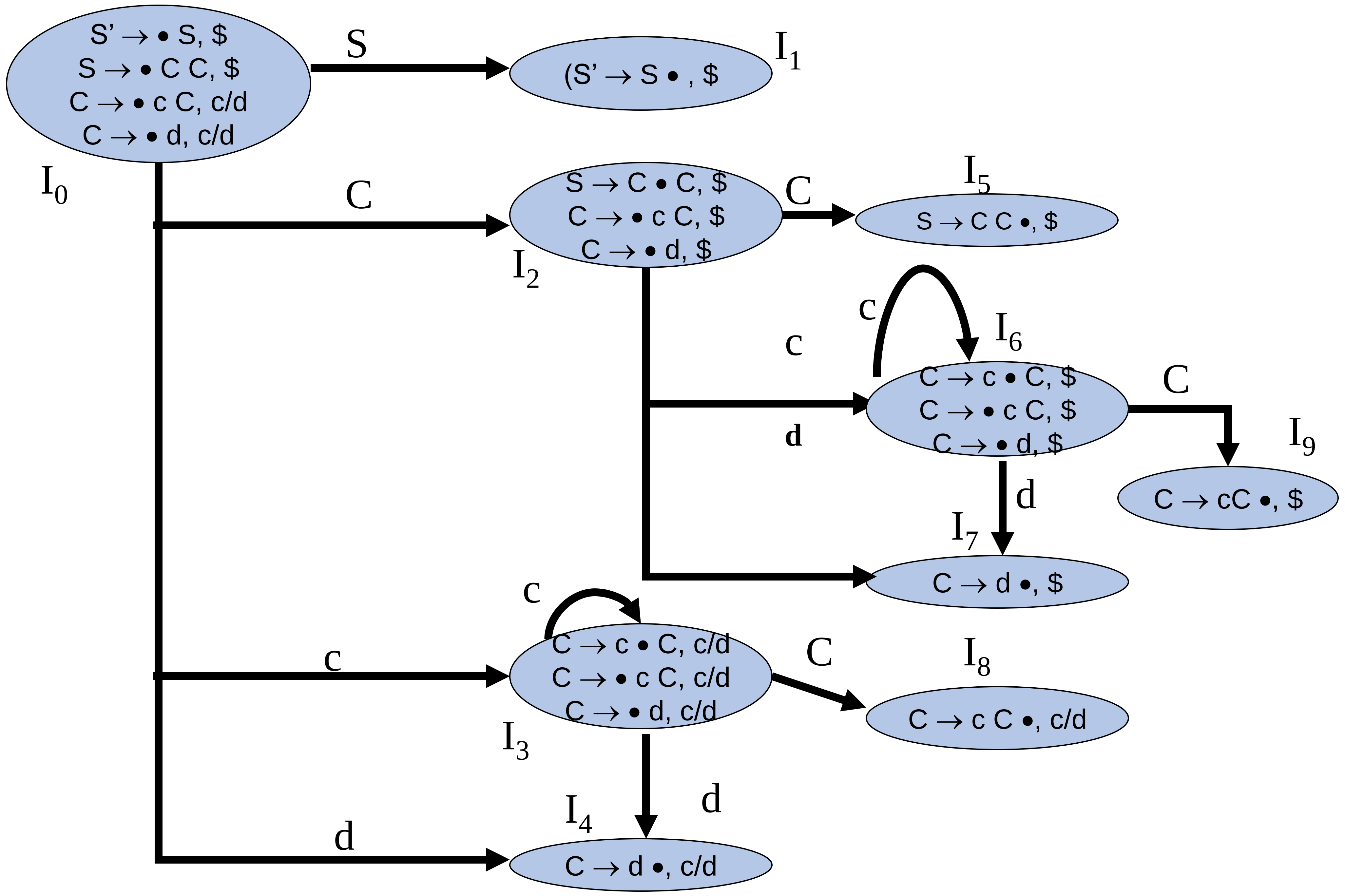
goto(I_3, c) = I_3

goto(I_3, d) = I_4

I_9: goto(I_6, C) =
$C \to cC\bullet, \$$

goto(I_6, c) = I_6

goto(I_6, d) = I_7



S’ → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
(S’ → S • , $
I1
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
c
I6
C → c • C, $
C → • c C, $
C → • d, $
d
C
I9
C → cC •, $
d
I7
C → d •, $
c
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
C
I8
C → c C •, c/d
d
I4
d
C → d •, c/d

Construction of LR(1) Parsing Tables

Algorithm: Construction of Canonical LR Parsing Tables
Input: An augmented grammar G'
Output: The canonical LR parsing table functions, action and goto for grammar G'

1. Construct C'= {I_0,...,I_n}, the canonical collection of sets of LR(1) items for G'.
2. State i of the parser is constructed from I_i. The parsing action for state i is determined as follows
   - If [$A\rightarrow\alpha.a\beta$,b] in I_i and goto(I_i,a)=I_j then set action[i,a] to ***shift j*.** Here *a* should be a terminal
   - If [$A\rightarrow\alpha.$,a] is in I_i and A≠S', then set action[i,a] to ***reduce*** $\boldsymbol{A\rightarrow\alpha}$
   - If [$S'\rightarrow S.$,\$] is in I_i , then action[i,\$] to ***accept***.

   If any conflicting actions generated by these rules, the grammar is not LR(1).
3. The goto transitions for state i are constructed for all nonterminals A using the rule:
   if goto(I_i,A)=I_j then goto[i,A]=j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser is the one constructed from the set of items containing [$S'\rightarrow .S$, \$]

Canonical LR(1) Collection- Example

$S' \rightarrow S$
1. $S \rightarrow CC$
2. $C \rightarrow cC$
3. $C \rightarrow d$

I_0: $S' \rightarrow \bullet S, \$$
$S \rightarrow \bullet CC, \$$
$C \rightarrow \bullet cC, c/)$
$C \rightarrow \bullet d, c/)$

I_1: goto(I_0, S)=
$S' \rightarrow S \bullet, \$$

I_2: goto(I_0, C) =
$S \rightarrow C \bullet C, \$$
$C \rightarrow \bullet cC, \$$
$C \rightarrow \bullet d, \$$

I_3: goto(I_0, c) =
$C \rightarrow c \bullet C, c/d$
$C \rightarrow \bullet cC, c/d$
$C \rightarrow \bullet d, c/d$

I_4: goto(I_0, d) =
$C \rightarrow d \bullet, c/d$

I_5: goto(I_2, C) =
$S \rightarrow CC \bullet, \$$

I_6: goto(I_2, c) =
$C \rightarrow c \bullet C, \$$
$C \rightarrow \bullet cC, \$$
$C \rightarrow \bullet d, \$$

I_7: goto(I_2, d) =
$C \rightarrow d \bullet, \$$

I_8: goto(I_3, C) =
$C \rightarrow cC \bullet, c/d$

goto(I_3, c) = I_3

goto(I_3, d) = I_4

I_9: goto(I_6, C) =
$C \rightarrow cC \bullet, \$$

goto(I_6, c) = I_6

goto(I_6, d) = I_7

Canonical LR(1) Parsing Table for Grammar G’

	ACTION			GOTO	
State	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		

goto(I_0,S)=I_1
goto(I_0,C)=I_2
goto(I_0,c)=I_3
goto(I_0,d)=I_4
goto(I_2,C)=I_5
goto(I_2,c)=I_6
goto(I_2,d)=I_7
goto(I_3,C)=I_8
goto(I_3,c)=I_3
goto(I_3,d)=I_4
goto(I_6,C)=I_9
goto(I_6,c)=I_6
goto(I_6,d)=I_7

LALR Parsing Tables

1. **LALR** stands for **Lookahead LR.**
2. LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
3. The number of states in SLR and LALR parsing tables for a grammar G are equal.
4. But LALR parsers recognize more grammars than SLR parsers.
5. ***yacc*** creates a LALR parser for the given grammar.
6. A state of LALR parser will be again a set of LR(1) items.

The Core of LR(1) Items

- The core of a set of LR(1) Items is the set of their first components (i.e., LR(0) items)
- The core of the set of LR(1) items

  { $C \rightarrow c\bullet C$, c/d,
  $C \rightarrow \bullet cC$, c/d,
  $C \rightarrow \bullet d$, c/d }

  is { $C \rightarrow c\bullet C$, $C \rightarrow \bullet cC$, $C \rightarrow \bullet d$ }

Creating LALR Parsing Tables

Canonical LR(1) Parser $\rightarrow$ LALR Parser

shrink # of states

- Look for LR(1) items having the same core, i.e. the first set of components
- Merge the sets with common core into one set of items
- Eg; I_4 and I_7 form such a pair with core {C→d.}
- I_3 and I_6 with core {C→c.C, C→.cC, C→.d}
- I_8 and I_9 with core {C→cC.}
- Replace I_4 and I_7 by I_{47}, [I_4 = C → d•, c/d ; I_7 = C → d•, $]
    - I_{47}={C→d., c/d/$}

Creation of LALR Parsing Tables

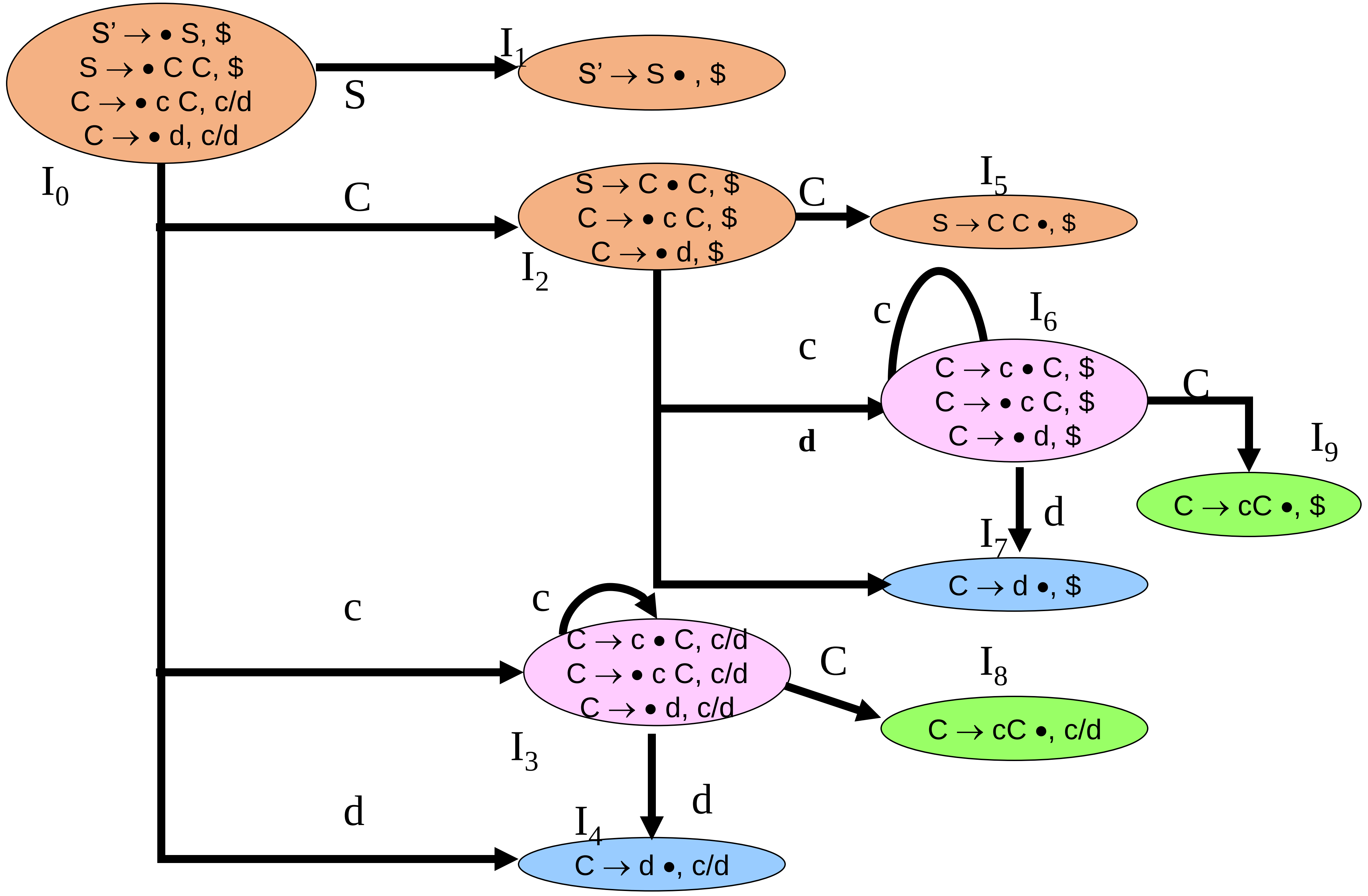
Algorithm: An easy but space consuming LALR parsing table construction

Input: An augmented grammar G’

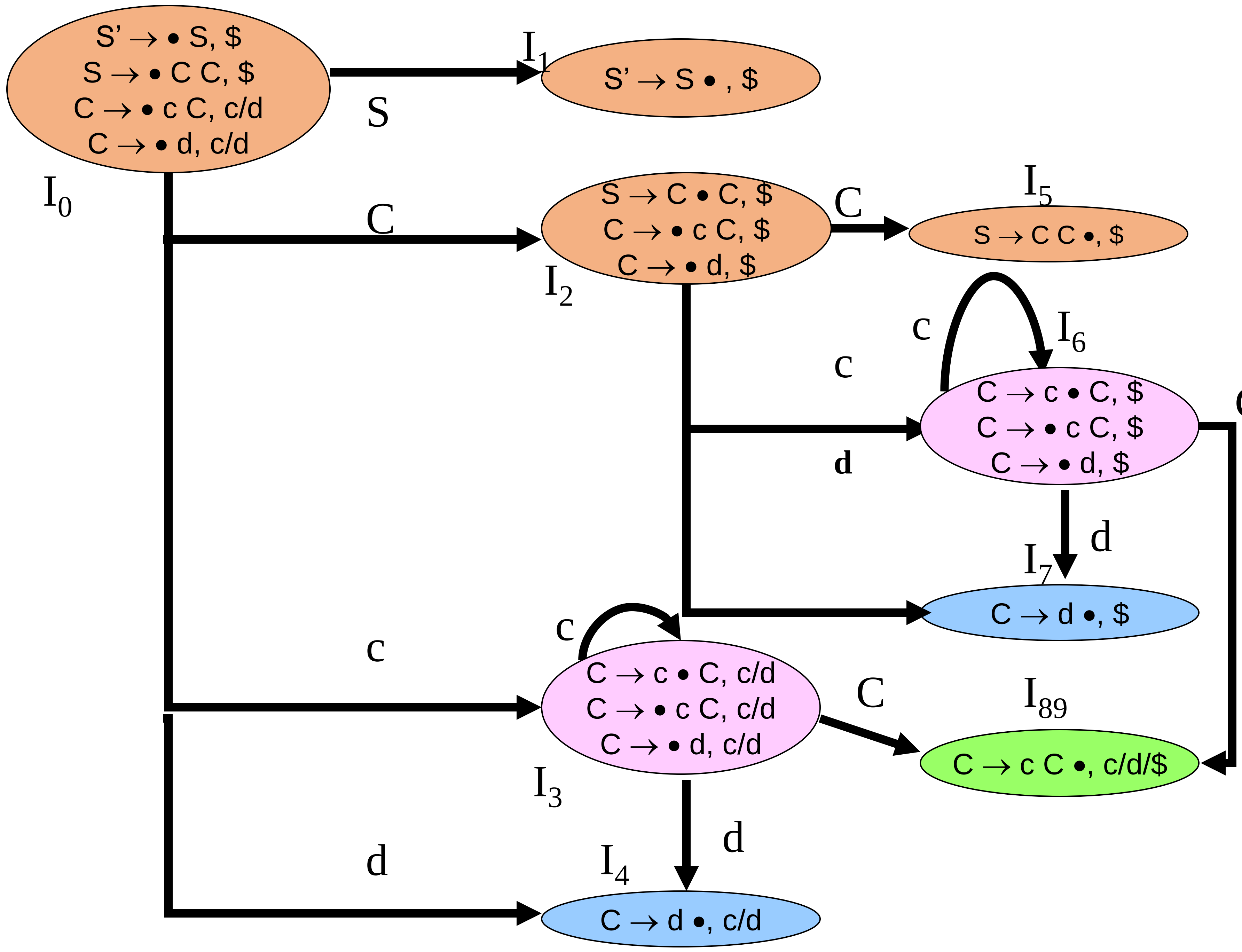
Output: The LALR paring table functions action and goto for grammar G’

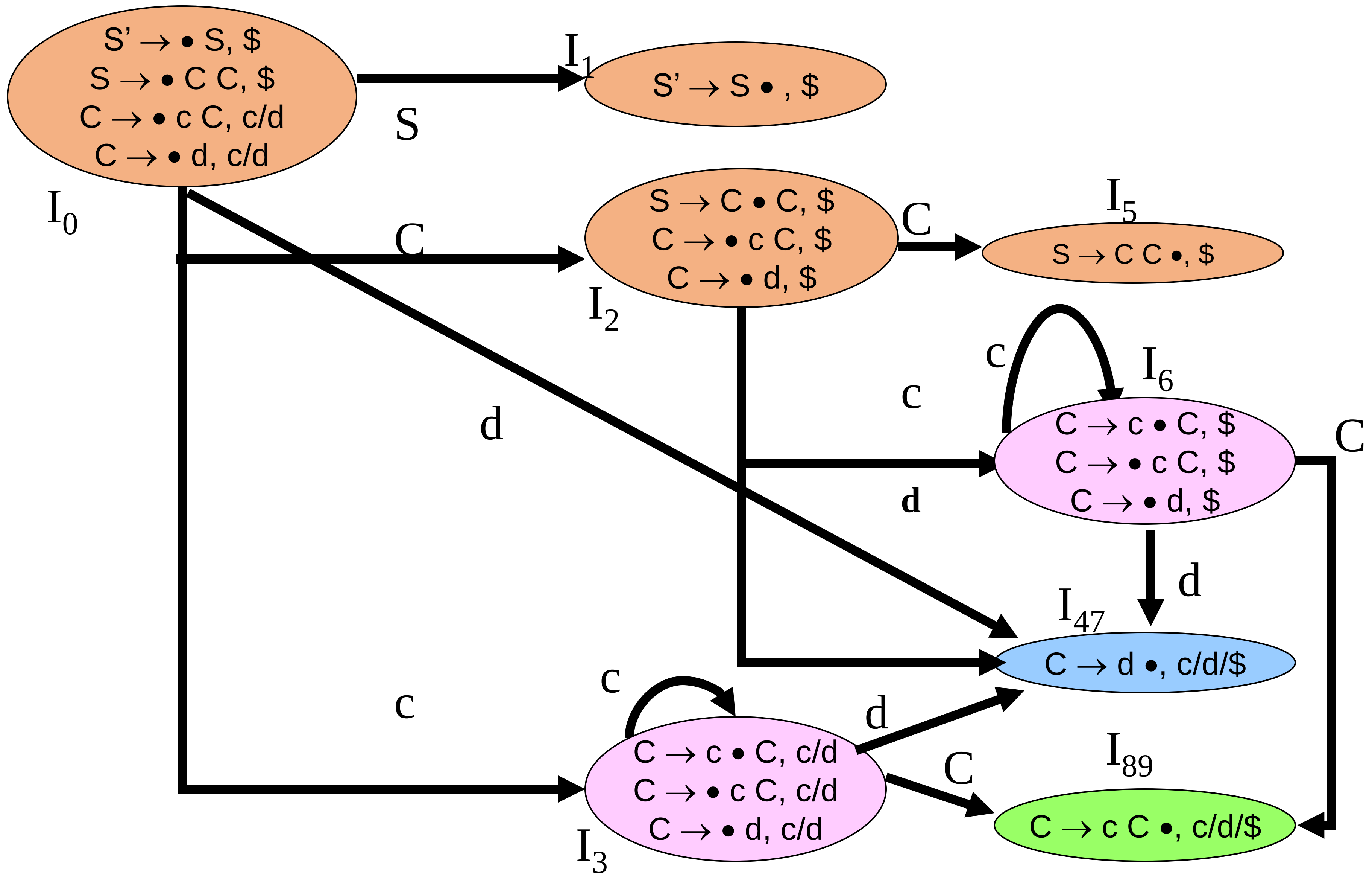
Method:

1. Construct C={I_0,...,I_n}, the collection LR(1) items for the given grammar.
2. For each core present among the sets of LR(1) items, find all sets having that core and replace these sets by their union.
3. Let C’={J_1,...,J_m} be the resulting set of LR(1) items. The parsing actions for state i are constructed from J_i in the same manner as in the algorithm for canonical LR parser. If there is a parsing action conflict, the algorithm fails to produce a parser and the grammar is not LALR(1).
4. The goto table is constructed as follows: If J is the union of one or more sets of LR(1) items, i.e. J=$I_1 \cup ... \cup I_{k,}$ then the cores of goto(I_1,X),...,goto(I_2,X) are the same since I_1,...,I_k have same cores. Let K be the union of all sets of items of goto(I,X). Then goto(I,X)=K.

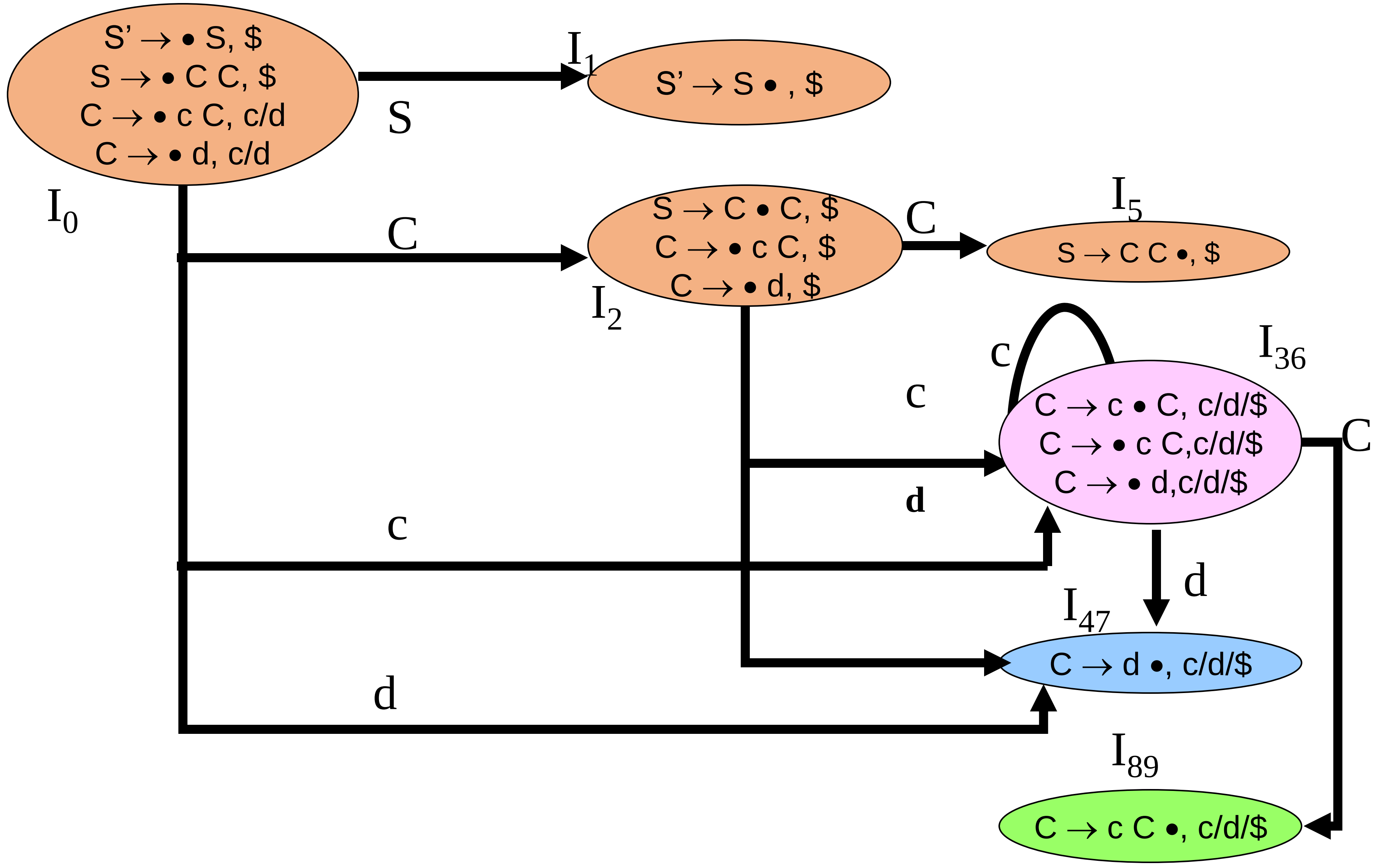


S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
c
I6
C → c • C, $
C → • c C, $
C → • d, $
d
C
I9
C → cC •, $
d
I7
C → d •, $
c
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
C
I8
C → cC •, c/d
d
d
I4
C → d •, c/d


S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
S' → S • , $
C
I2
S → C • C, $
C → • c C, $
C → • d, $
C
I5
S → C C •, $
c
c
I6
C → c • C, $
C → • c C, $
C → • d, $
d
C
d
I7
C → d •, $
c
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
C
I89
C → c C •, c/d/$
d
d
I4
C → d •, c/d



I_0
S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I_1
S' → S • , $
S
C
I_2
S → C • C, $
C → • c C, $
C → • d, $
C
I_5
S → C C •, $
d
c
I_6
C → c • C, $
C → • c C, $
C → • d, $
c
d
C
d
I_{47}
C → d •, c/d/$
c
c
I_3
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
d
C
I_{89}
C → c C •, c/d/$



I_0
S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
S
I_1
S' → S • , $
C
I_2
S → C • C, $
C → • c C, $
C → • d, $
C
I_5
S → C C •, $
c
c
I_{36}
C → c • C, c/d/$
C → • c C,c/d/$
C → • d,c/d/$
d
c
C
d
I_{47}
C → d •, c/d/$
d
I_{89}
C → c C •, c/d/$

LALR Parsing Table

States	action			goto	
	c	**d**	**$**	**S**	**C**
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

- r2

LALR Parsing Table

States	**action**			**goto**	
	c	**d**	**$**	**S**	**C**
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Canonical LR(1) Parsing Table

	ACTION			GOTO	
State	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		