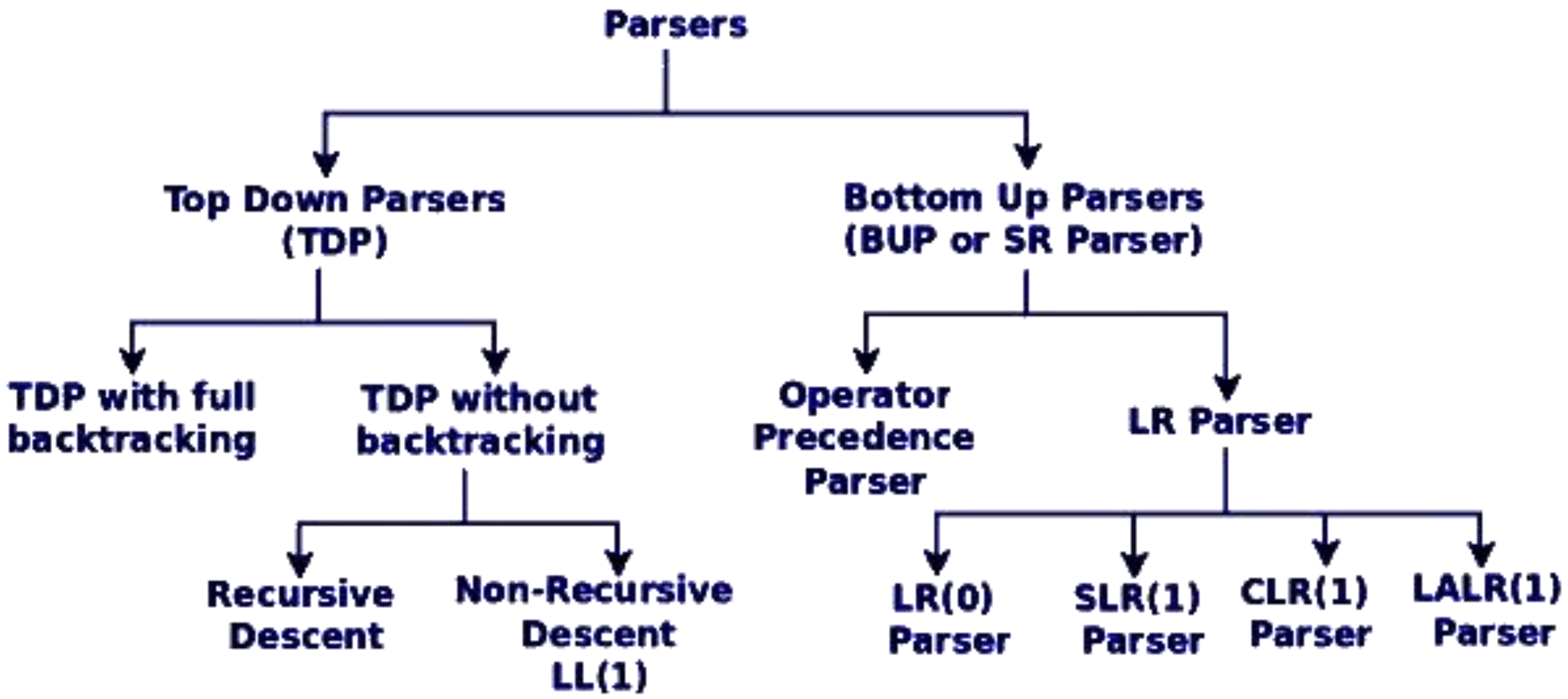
COMPILER DESIGN

HANDLED BY
DIVYA B
DEPT. OF CSE
VJEC, CHEMPERI


Parsers
Top Down Parsers (TDP)
Bottom Up Parsers (BUP or SR Parser)
TDP with full backtracking
TDP without backtracking
Operator Precedence Parser
LR Parser
Recursive Descent
Non-Recursive Descent LL(1)
LR(0) Parser
SLR(1) Parser
CLR(1) Parser
LALR(1) Parser

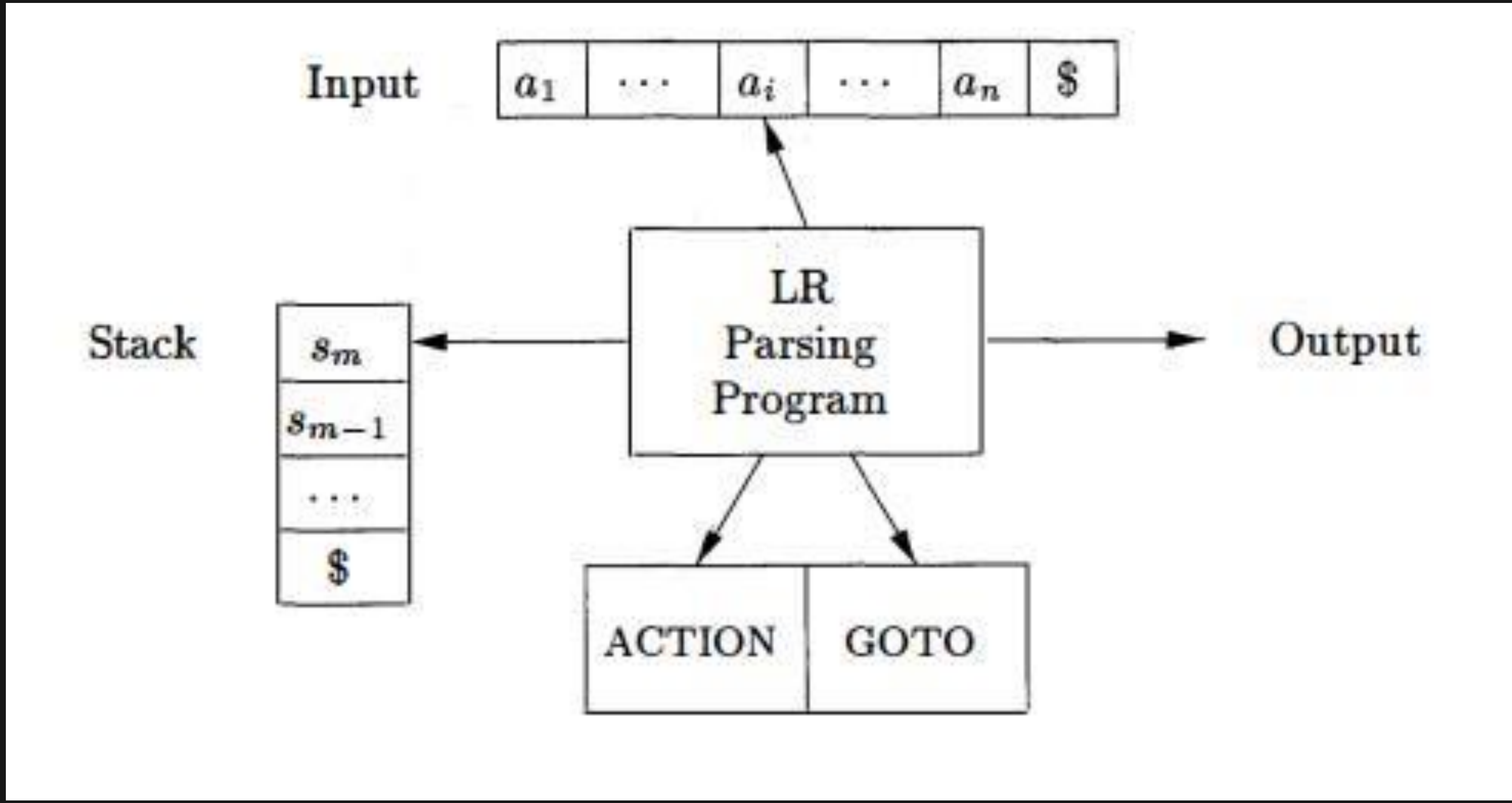
LR PARSING

- LR parsing is most efficient method of bottom up parsing which can be used to parse large class of context free grammar.
- The technique is called LR(k) parsing
  - “L” is for left to right scanning of input symbol.
  - “R” is for constructing right most derivation in reverse.
  - k denotes the number of input symbols for lookahead that are used in making the parsing decision.

LR PARSING

- There are 3 types of LR parsing
  - SLR (Simple LR)
  - CLR (Canonical LR)
  - LALR (Look Ahead LR)

Model of a LR Parser



LR PARSING

- The LR parsing program (driver program) is the same for all LR parsers.
- Only the parsing table changes from one parser to another.
- The parsing program reads characters from an input buffer one at a time.
- The shift reduce parser shifts a symbol whereas the LR parser shifts a state.
- Each state summarizes the information contained in the stack below it.

LR PARSING

- ***Input Buffer***
  - The parsing program reads characters from an input buffer one at a time.

LR PARSING

- **Stack**
  - The stack holds a sequence of states $\mathbf{s_0 s_1 s_2 \ldots s_m}$ where s_m is on the top.
  - Combination of state symbol on stack top and current input symbol are used to index the parsing table and determine the shift reduce parsing decision.

LR PARSING

- *Stack*
  - States correspond to set of items, and there is a transition from state i to state j if GOTO(i, X) = j.
  - All transitions to state j must be for the same grammar symbol X.
  - Each state, except the start state 0, has a unique grammar symbol associated with it.

Structure Of The LR Parsing Table

- Consists of two parts
  - **ACTION** function
  - **GOTO** function

ACTION

- This function takes as arguments a state *i* and a terminal a (or $, the input end marker).
- The value of ACTION [i, a] can have one of the four forms

1. Shift *j*, where *j* is a state. The action taken by the parser effectively shifts input a to the stack, but uses state j to represent a.
2. Reduce $A \rightarrow \beta$. The action of the parser effectively reduces β on the top of the stack to head A.
3. Accept
4. Error

- **GOTO**
  - This function takes a state and grammar symbol as arguments and produces a state.
  - If GOTO [i, A] = j then GOTO maps a state i and a nonterminal A to state j.

LR Parser Configurations

- The configuration of LR parser is a pair

$$(s_0 s_1 s_2 \ldots s_m ,\ a_i a_{i+1} \ldots a_n \$)$$

The first component is the stack contents (top on the right), and the second component is the remaining input.

LR Parser Configurations

- This configuration represents the right sentential form

  $$X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n$$

  in essentially the same way as a shift reduce parser would; the only difference is that instead of grammar symbols, the stack holds states from which grammar symbols can be recovered.

LR Parser Configurations

- X_i is the grammar symbol represented by state s_i.
- s_0 the start state of the parser does not represent a grammar symbol. It serves as the bottom-of-stack marker.

Working Of LR Parser

- The configuration resulting after each type of move are as follows
  1. If ACTION[s_m, a_i] = shift s, the parser executes a shift move; it shifts the next state s onto the stack, entering the configuration

$$(s_0 s_1 s_2 \ldots s_m s,\ a_{i+1} \ldots a_n \$)$$

The current input symbol is now a_{i+1}

2. If ACTION[s_m, a_i] = reduce $A \rightarrow \beta$, then the parser executes a reduce move, entering the configuration

($s_0 s_1 \ldots s_{m-r} s$, $a_i a_{i+1} \ldots a_n$ \$)

where r is the length of β and s = GOTO[s_{m-r}, A]

Here the parser first popped r state symbols off the stack, exposing the state **s_{m-r}** .

The parser then pushed s, the entry for GOTO[s_{m-r} , A] onto the stack.

The current input symbol is not changed in a reduce move.

3. If ACTION $[s_m, a_i]$ = accept, parsing is completed.
4. If ACTION $[s_m, a_i]$ = error, the parser has discovered an error and it calls an error recovery routine.

LR Parser Algorithm

- INPUT : An input string w and a LR parsing table with functions ACTION and GOTO for a grammar G.
- OUTPUT : If w is in L(G), the reduction steps for a bottom up parse for w; otherwise an error indication.
- METHOD : Initially, the parser has s_0 on its stack, where s_0 is the initial state and w$ in the input buffer.

```
let a be the first symbol of w$;
while(1) {
        let s be the state on top of the stack;
        if(ACTION[s,a] = shift t) {



        } else if(ACTION[s, a] = reduce A→ β) {





        } else if(ACTION[s,a]=accept} break;
        else call error-recovery routine;
}
```

- Consider the expression grammar

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

- Show the LR parser moves on the input string id*id+id

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			

	Stack	Symbols	Input	Action
(1)	0		$\mathbf{id} * \mathbf{id} + \mathbf{id}\,\$$	shift
(2)	0 5	$\mathbf{id}$	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $F \to \mathbf{id}$
(3)	0 3	F	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $T \to F$
(4)	0 2	T	$* \mathbf{id} + \mathbf{id}\,\$$	shift
(5)	0 2 7	$T *$	$\mathbf{id} + \mathbf{id}\,\$$	shift
(6)	0 2 7 5	$T * \mathbf{id}$	$+ \mathbf{id}\,\$$	reduce by $F \to \mathbf{id}$
(7)	0 2 7 10	$T * F$	$+ \mathbf{id}\,\$$	reduce by $T \to T * F$
(8)	0 2	T	$+ \mathbf{id}\,\$$	reduce by $E \to T$
(9)	0 1	E	$+ \mathbf{id}\,\$$	shift
(10)	0 1 6	$E +$	$\mathbf{id}\,\$$	shift
(11)	0 1 6 5	$E + \mathbf{id}$	$\$$	reduce by $F \to \mathbf{id}$
(12)	0 1 6 3	$E + F$	$\$$	reduce by $T \to F$
(13)	0 1 6 9	$E + T$	$\$$	reduce by $E \to E + T$
(14)	0 1	E	$\$$	accept

CONSTRUCTION OF SLR PARSING TABLE

- **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.
  - An LR(0) item (item) of a grammar G is a production of G with a dot at the some position of the right side.

CONSTRUCTION OF SLR PARSING TABLE

- **LR(0) ITEM**
  - E.g. for the productions A → aBb possible LR(0) items are

$$A \rightarrow .aBb$$

$$A \rightarrow a.Bb$$

$$A \rightarrow aB.b$$

$$A \rightarrow aBb.$$

CONSTRUCTION OF SLR PARSING TABLE

- **LR(0) ITEM**
  - A production rule of the form A→ε yields only one item A→ .
  - Intuitively, an item shows how much of a production we have seen till the current point in the parsing procedure.
  - A collection of sets of LR(0) items (the canonical LR(0) collection) is the basis for constructing SLR parsers.

CONSTRUCTION OF SLR PARSING TABLE

- **LR(0) ITEM**
  - To construct the canonical LR(0) collection for a grammar we define
    - Augmented grammar
    - Two functions – CLOSURE and GOTO

CONSTRUCTION OF SLR PARSING TABLE

- **AUGMENTED GRAMMAR**
  - G′ is the augmented grammar of G with a new production rule S′→S where S′ is the new starting symbol

$$G' = G \cup \{S' \rightarrow S\} \text{ where S is the start state of G.}$$

  - This is done to signal to the parser when the parsing should stop to announce acceptance of input.

CONSTRUCTION OF SLR PARSING TABLE

- **KERNEL AND NON-KERNEL ITEMS**
  - Kernel items include the set of items that do not have the dot at leftmost end.
  - S′→ .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.

CONSTRUCTION OF SLR PARSING TABLE

- **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 $A \rightarrow \alpha.B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow .\gamma$ will be in the closure(I). We will apply this rule until no more new LR(0) items can be added to closure(I).

```
function closure ( I )
        begin
        J := I;
                repeat
                for each item A → α .Bβ in J and each production
                        B→γ of G such that B→.γ is not in J do
                                add B→.γ to J
                until no more items can be added to J
            return J
        end
```

CONSTRUCTION OF SLR PARSING TABLE

- **CLOSURE OPERATION**

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

CLOSURE({[E'→E]})

{ $E' \rightarrow \bullet E$ ← kernel items
$E \rightarrow \bullet E+T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet T*F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet (E)$
$F \rightarrow \bullet id$ }

CONSTRUCTION OF SLR PARSING TABLE

- **GOTO FUNCTION**
  - 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 to be closure of the set of all items [$A \rightarrow \alpha X.\beta$] such that [$A \rightarrow \alpha .X\beta$] is in I.
  - The GOTO function is used to define the transitions in the LR(0) automaton for a grammar.

CONSTRUCTION OF SLR PARSING TABLE

- **GOTO FUNCTION**
  - The states of the automaton correspond to set of items.
  - GOTO(I,X) specifies the transition from the state for I under input X.

CONSTRUCTION OF SLR PARSING TABLE

- **GOTO FUNCTION**
  - If I is the set of two items {[E'→E.], [E→E.+T]} the GOTO(I,+) contains the items

E→E+.T
T→.T*F
T→.F
F→.(E)
F→.id

GOTO(I,+) is computed by examining I for items with + immediately to the right of the dot. Then the dot is moved over + to get E→E+.T and the closure of this set is taken.

CONSTRUCTION OF SLR PARSING TABLE

CONSTRUCTION OF CANONICAL LR(0) COLLECTION

To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.

CONSTRUCTION OF CANONICAL LR(0) COLLECTION

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

CANONICAL COLLECTION OF LR(0) ITEMS

1. Augment the grammar
2. Draw canonical collection of LR(0) items
3. Number the production
4. Create the parsing table
5. Stack implementation
6. Draw parse tree

CANONICAL COLLECTION OF LR(0) ITEMS

- Consider the input string as ccdd and the productions as

  E→BB

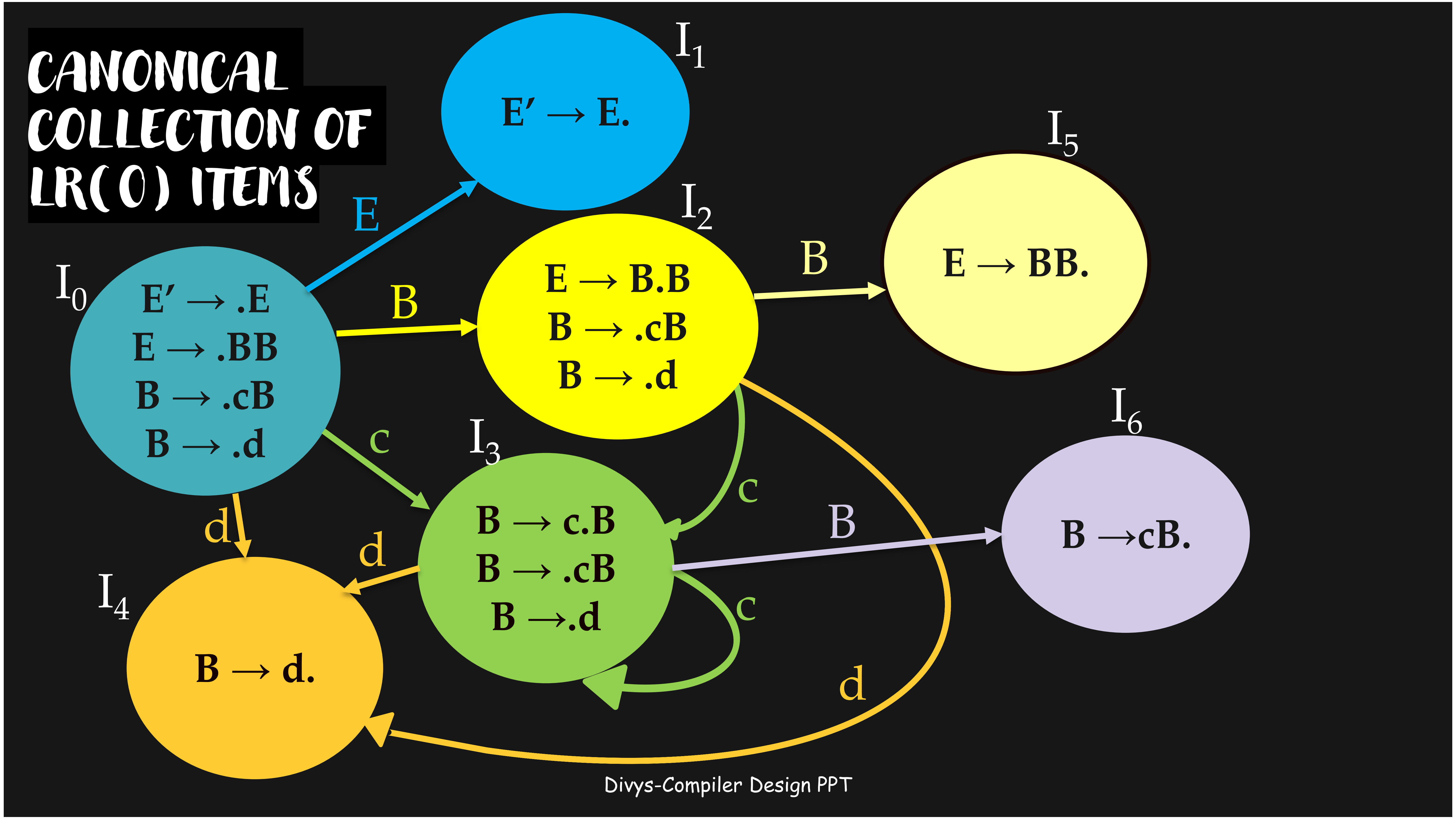
  B→cB/d

Step 1: Augment the given grammar

E'→E

E→BB

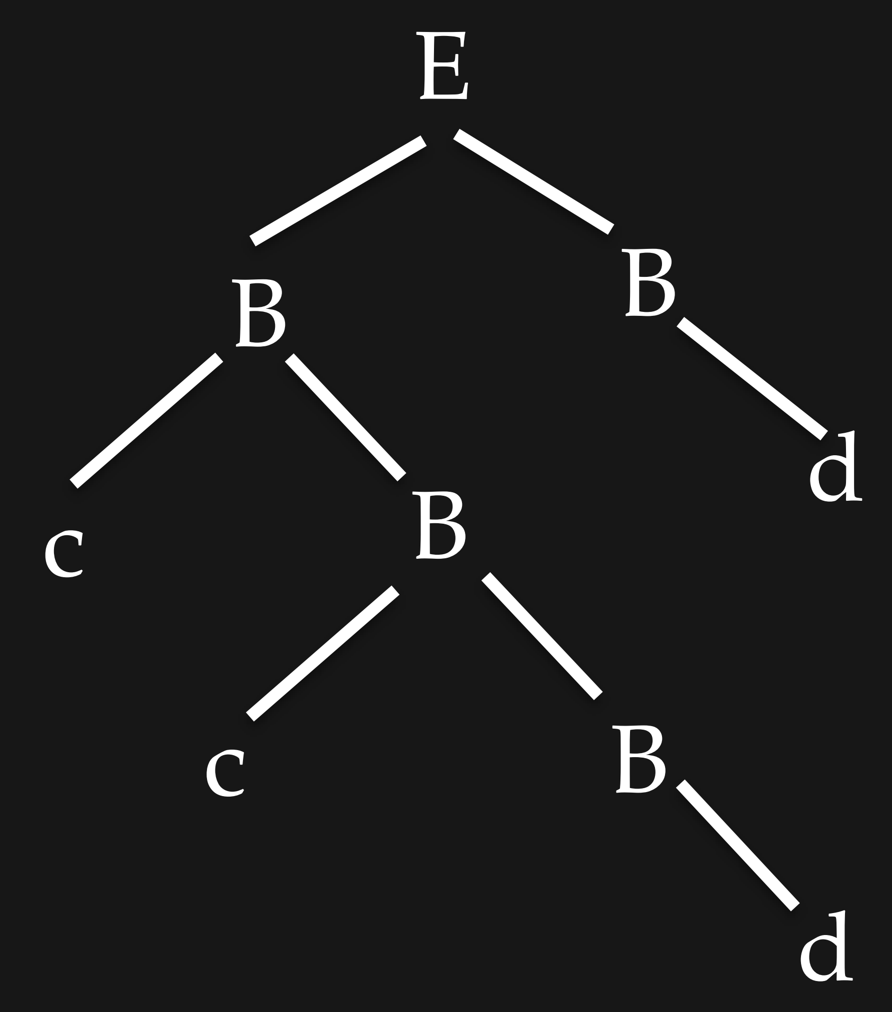
B→cB/d


CANONICAL COLLECTION OF LR(0) ITEMS
I0
E′ → .E
E → .BB
B → .cB
B → .d
E
I1
E′ → E.
B
I2
E → B.B
B → .cB
B → .d
B
I5
E → BB.
c
I3
B → c.B
B → .cB
B →.d
c
B
I6
B →cB.
c
d
d
I4
B → d.
d
Divys-Compiler Design PPT

LR(0) PARSING TABLE

Stack	ACTION			GOTO	
	c	d	$	E	B
0	s3	s4		1	2
1			Accept		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

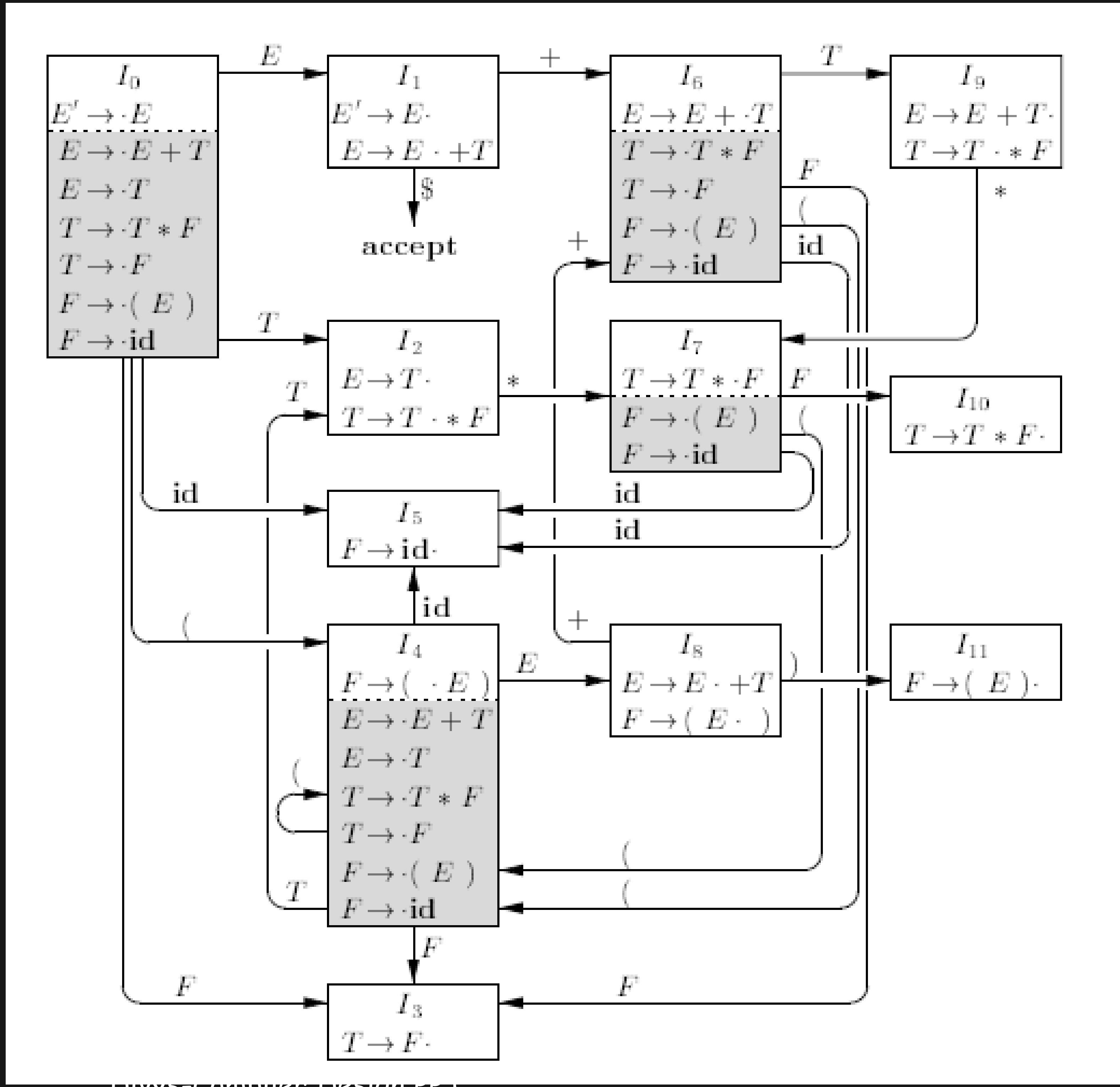
Stack	Symbol	Input	Action
0		ccdd$	Shift
03	c	cdd$	Shift
033	cc	dd$	Shift
0334	ccd	d$	Reduce by B→d
0336	ccB	d$	Reduce by B→cB
036	cB	d$	Reduce by B→cB
02	B	d$	Shift
024	Bd	$	Reduce by B→d
025	BB	$	Reduce by B→BB
01	E	$	Accept



Divys-Compiler Design PPT

CONSTRUCTING SLR PARSING TABLES

E′→E
1. E→E+T
2. E→T
3. T→T*F
4. T→F
5. F→(E)
6. F→id



LR(0) PARSING TABLE

Stack	ACTION						GOTO		
	id	**+**	*****	**(**	**)**	**$**	E	T	F
0	**s5**			**s4**			**1**	**2**	**3**
1		**s6**				**Acc**			
2	**r2**	**r2**	**s7/r2**	**r2**	**r2**	**r2**			
3									
4									
5									
6									
7									
8									
9									
10									
11									

Shift-reduce conflict

CONSTRUCTING SLR PARSING TABLE

INPUT: An augmented grammar G'.

OUTPUT: The SLR parsing table functions *action* and *goto* for G' .

METHOD:

1. Construct C ={$I_0, I_1, \ldots..I_n$}, the collection of sets of LR(0) items for G'.
2. State *i* is constructed from I_i. The parsing actions for state i are determined as follows:

   (a) If [$A \to \alpha \cdot a\beta$] is in I_i and goto(I_i,*a*) = I_j, then set *action*[*i*,*a*] to "shift j". Here *a* must be terminal.

   (b) If [$A \to \alpha \cdot$] is in I_i , then set *action*[*i*,*a*] to "reduce $A \to \alpha$" for all *a* in FOLLOW(A).

   (c) If [$S' \to S.$] is in I_i, then set *action*[*i*,$] to "accept".

CONSTRUCTING SLR PARSING TABLE

If any conflicting actions are generated by the above rules, we say the grammar is not SLR (1). The algorithm fails to produce a parser in this case.

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 rules (2) and (3) are made "error".
5. The initial state of the parser is the one constructed from the set of items containing [S′ → S].

CONSTRUCTING SLR PARSING TABLE

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

- E.g. for state I_2 the E→T. is a final state and so we need to check the production number which is 2. Then we need to r2 in the columns corresponding to FOLLOW(E).

SLR PARSING TABLE

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			

VIABLE PREFIXES

- The LR(0) automaton for a grammar characterizes the strings of grammar symbols that can appear on the stack of a shift-reduce parser for the grammar.
- The stack contents must be a prefix of a right sentential form.
- If the stack holds α and the rest of the input is x, then a sequence of reductions will take αx to S.
- In terms of derivations, $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha x$

VIABLE PREFIXES

- The prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes.
- It is always possible to add terminal symbols to the end of a viable prefix to obtain a right sentential form.

VIABLE PREFIXES

- SLR parsing is based on the fact that LR(0) automata recognize viable prefixes.
- Item $A \rightarrow \beta_1 . \beta_2$ is valid for a viable prefix $\alpha\beta_1$ if there is a derivation S′$\underset{rm}{\overset{*}{\Rightarrow}}$ αAw$\underset{rm}{\Rightarrow}$$\alpha\ \beta_1\ \beta_2$w.
- An item will be valid for many viable prefixes.

More Powerful LR Parsers

- The **"canonical-LR"** or just "LR" method makes full use of the lookahead symbol(s).
- This method uses a large set of items, called the LR(1) items.
- A LR(1) item is $A \rightarrow \alpha.\beta, a$ where a is the lookahead.

  a is a terminal or end marker.

ALGORITHM FOR CONSTRUCTION OF CANONICAL LR PARSING TABLES

INPUT : An augmented grammar G’

OUTPUT : The canonical-LR parsing table functions ACTION and GOTO for G'.

1. Construct $C' = \{I_0, I_1, \cdots, I_n\}$, the 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.

   (a) If $[A \to \alpha \cdot a\beta, b]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to "shift j." Here a must be a terminal.

   (b) If $[A \to \alpha \cdot, a]$ is in I_i, $A \neq S'$, then set $\text{ACTION}[i, a]$ to "reduce $A \to \alpha$."

   (c) If $[S' \to S \cdot, \$]$ is in I_i, then set $\text{ACTION}[i, \$]$ to "accept."

   If any conflicting actions result from the above rules, we say the grammar is not LR(1). The algorithm fails to produce a parser in this case.

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

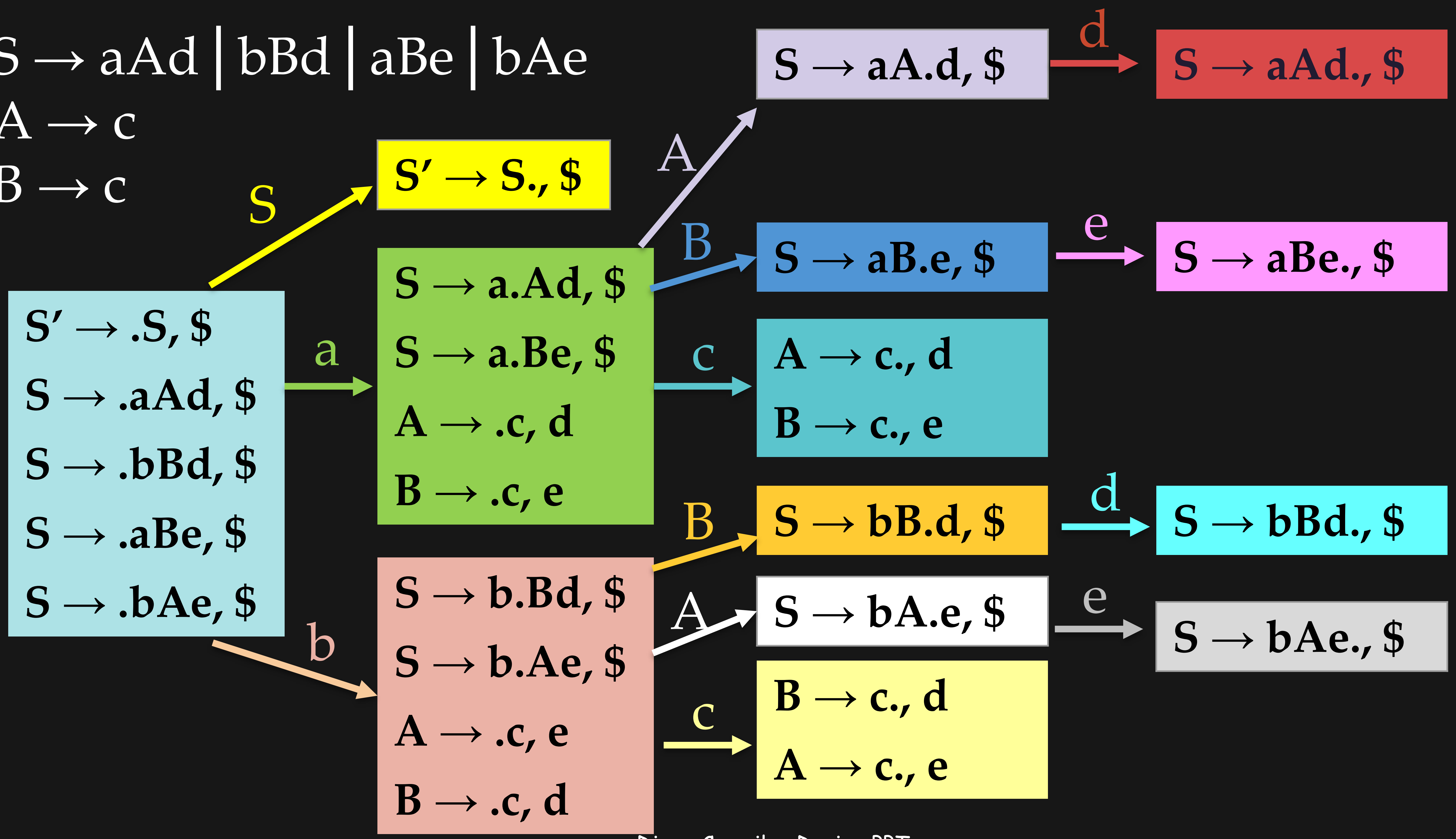
4. All entries not defined by rules (2) and (3) are made "error."

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

$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$

$A \rightarrow c$

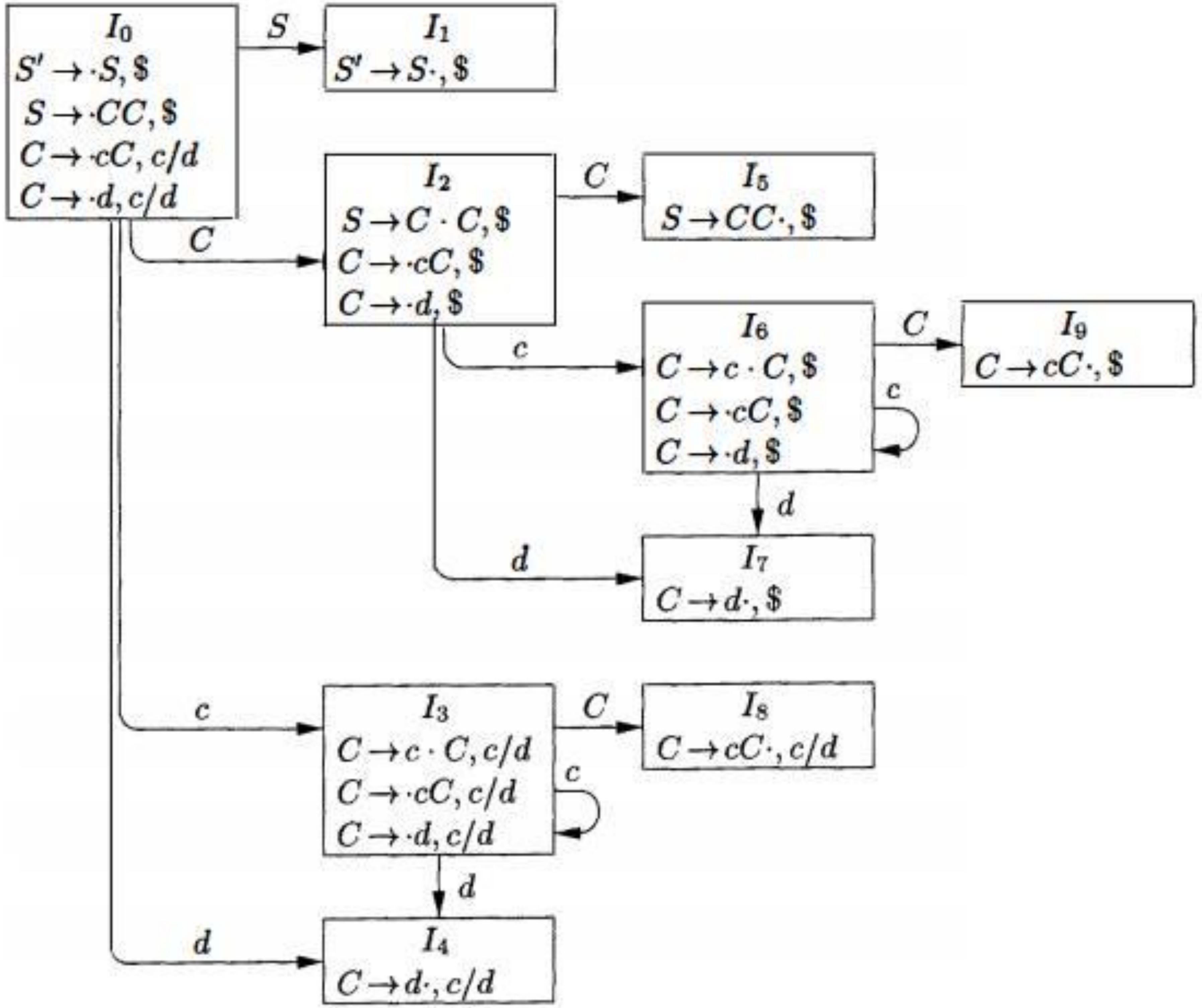
$B \rightarrow c$



Stack	ACTION						GOTO		
	a	b	c	d	e	$	S	A	B
0	s2	s3					1		
1						Accept			
2			s6					4	5
3			s9					8	7
4				s10					
5					s11				
6				r5	r6				
7				s12					
8					s13				
9				r6	r5				
10						r1			
11						r3			
12						r2			
13						r4			

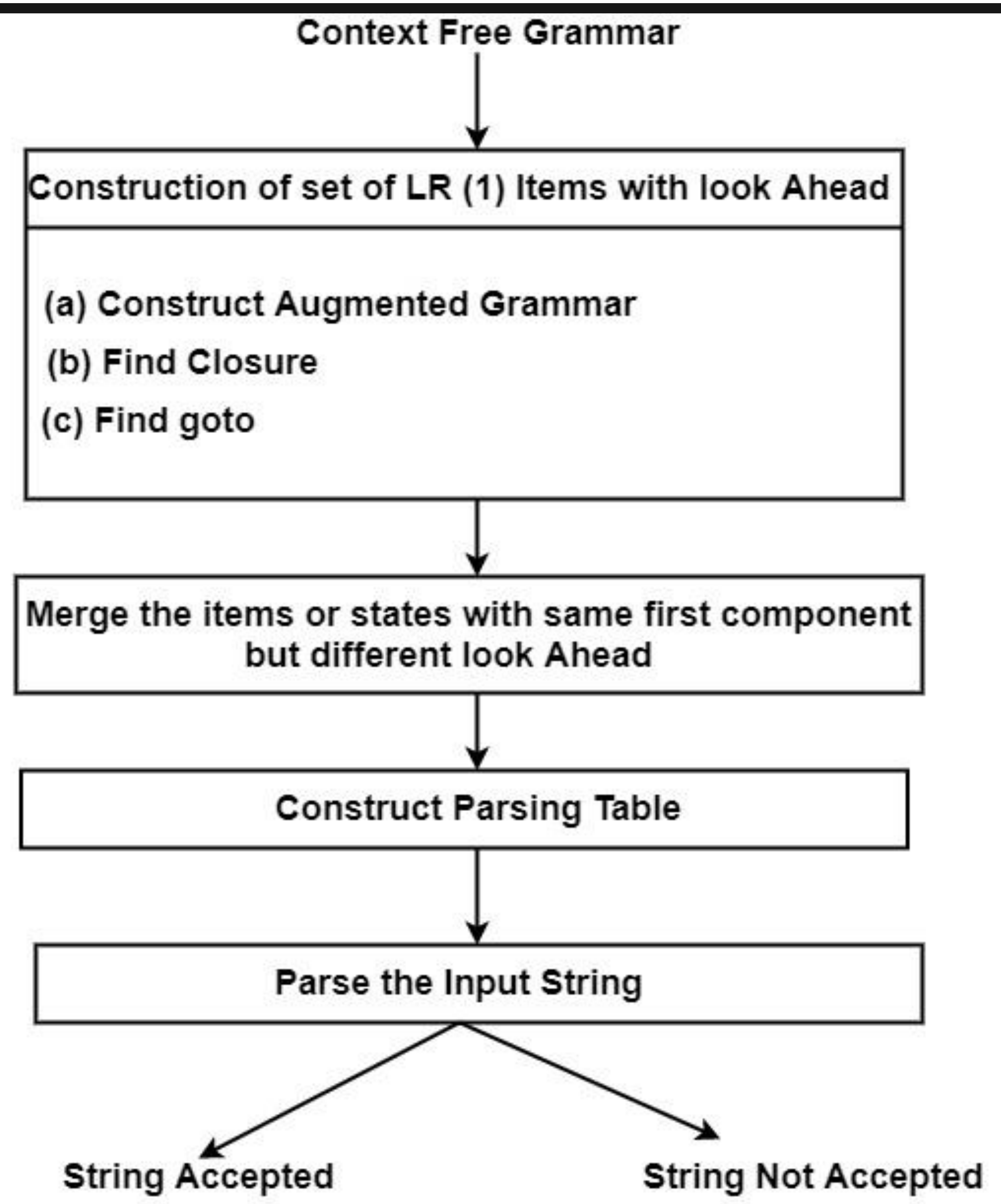
$S \rightarrow CC$

$C \rightarrow cC \mid d$



More Powerful LR Parsers

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



Working of LALR Parser

INPUT: An augmented grammar G′.

OUTPUT: The LALR parsing – table functions ACTION and GOTO for G′

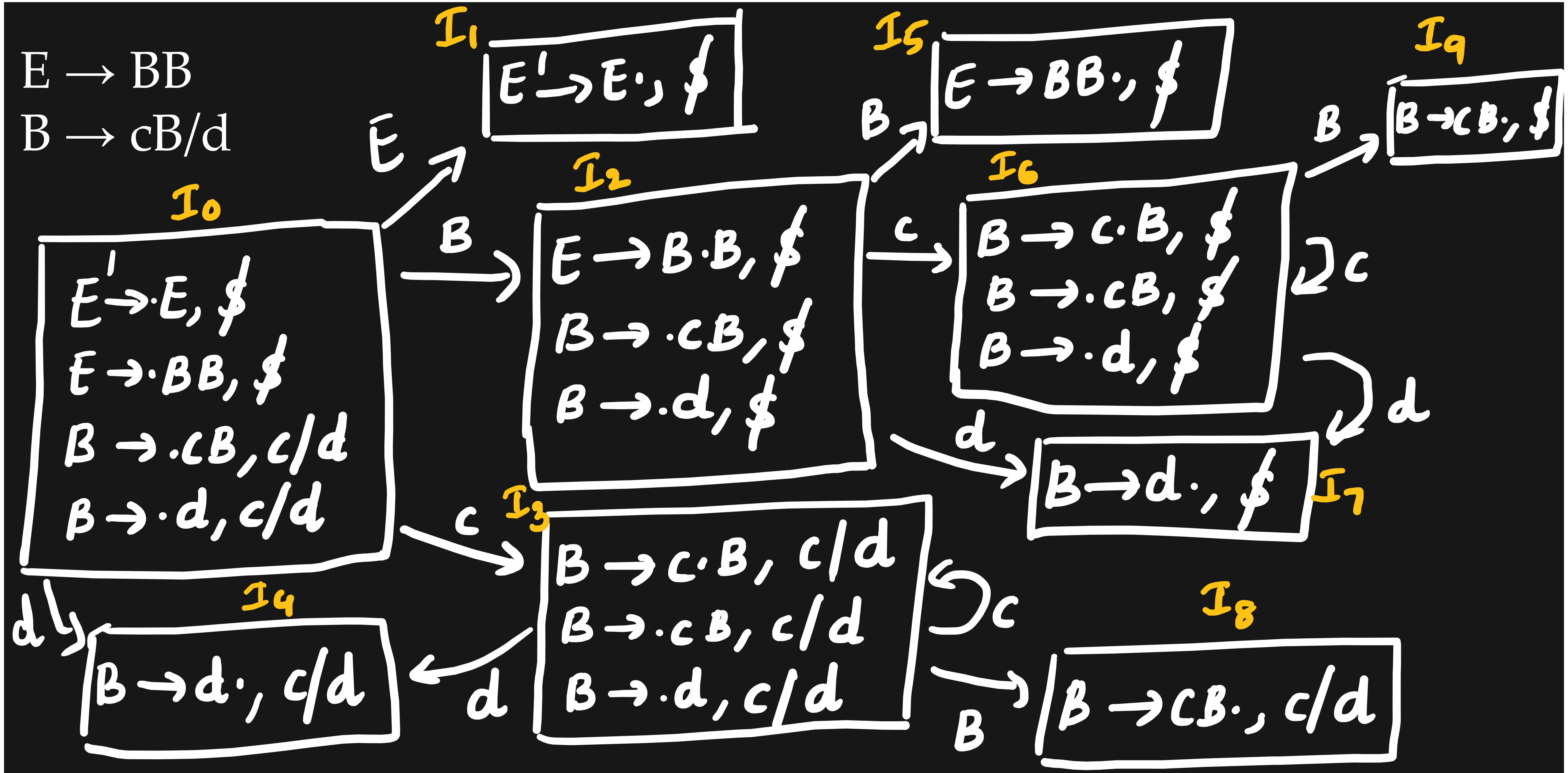
METHOD:

1. Construct C = { $I_0, I_1, \ldots, I_n$ }, the collection of sets of LR(1) items.
2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.
3. Let C′ = {$J_0, J_1, \ldots, J_m$}, the collection of sets of LR(1) items. The parsing action for state i are constructed from J_i in the same manner as in Algorithm. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1)
4. The GOTO table is constructed as follows. if J is the union of one or more sets of LR(1) items, that is, $J = I_1 \cup I_2 \cup \ldots \cup I_k$, then the $I_1, I_2, \ldots, I_k$ all have the same core. Let K be the union of all sets of items having the same core as GOTO(I_1,X). Then GOTO (J, X) = K.

$E \rightarrow BB$
$B \rightarrow cB/d$

$E \rightarrow BB$

$B \rightarrow cB/d$



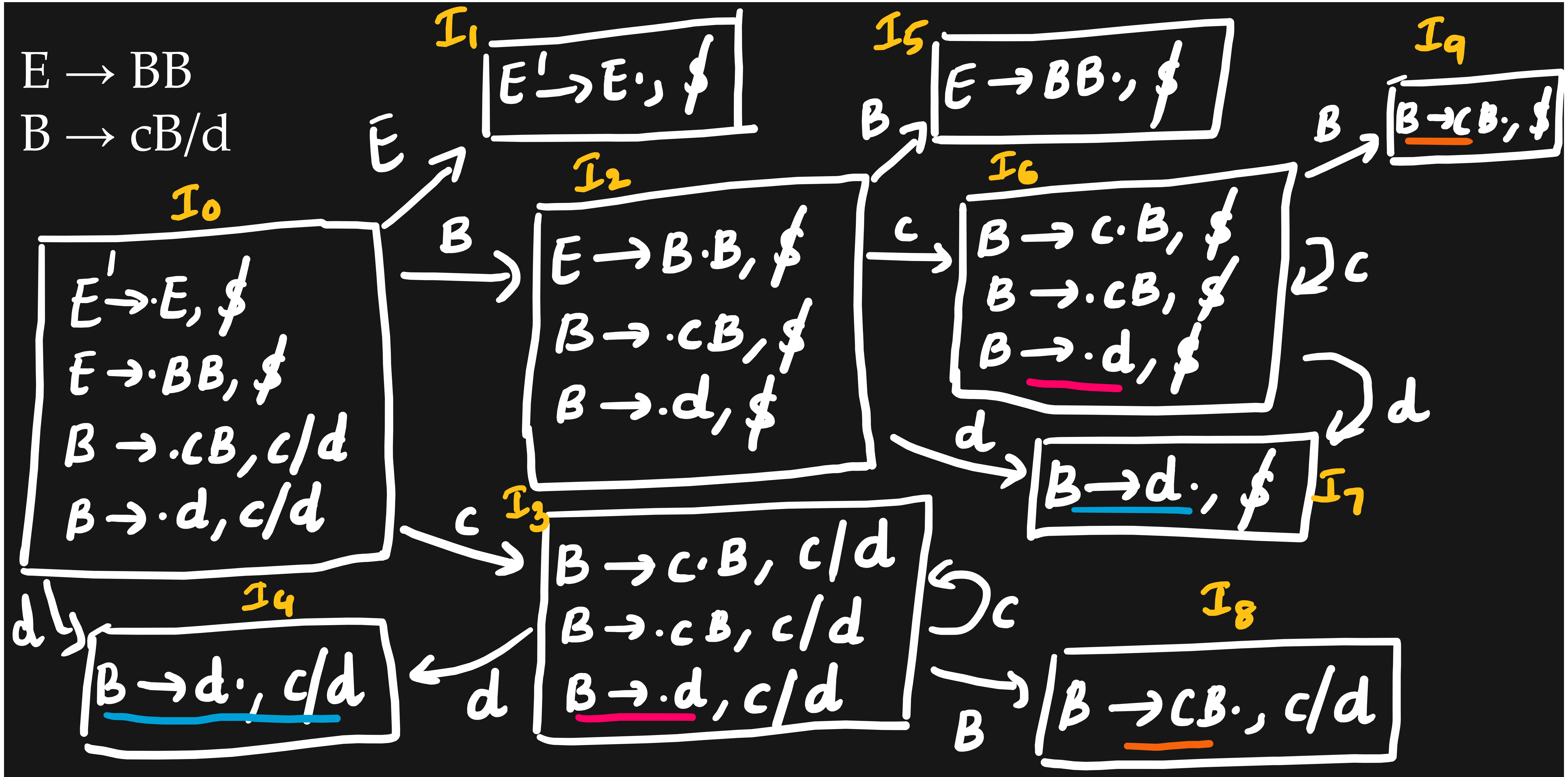
Divys-Compiler Design PPT

CLR PARSING TABLE

State	ACTION			GOTO	
	c	d	$	E	B
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		

$E \rightarrow BB$

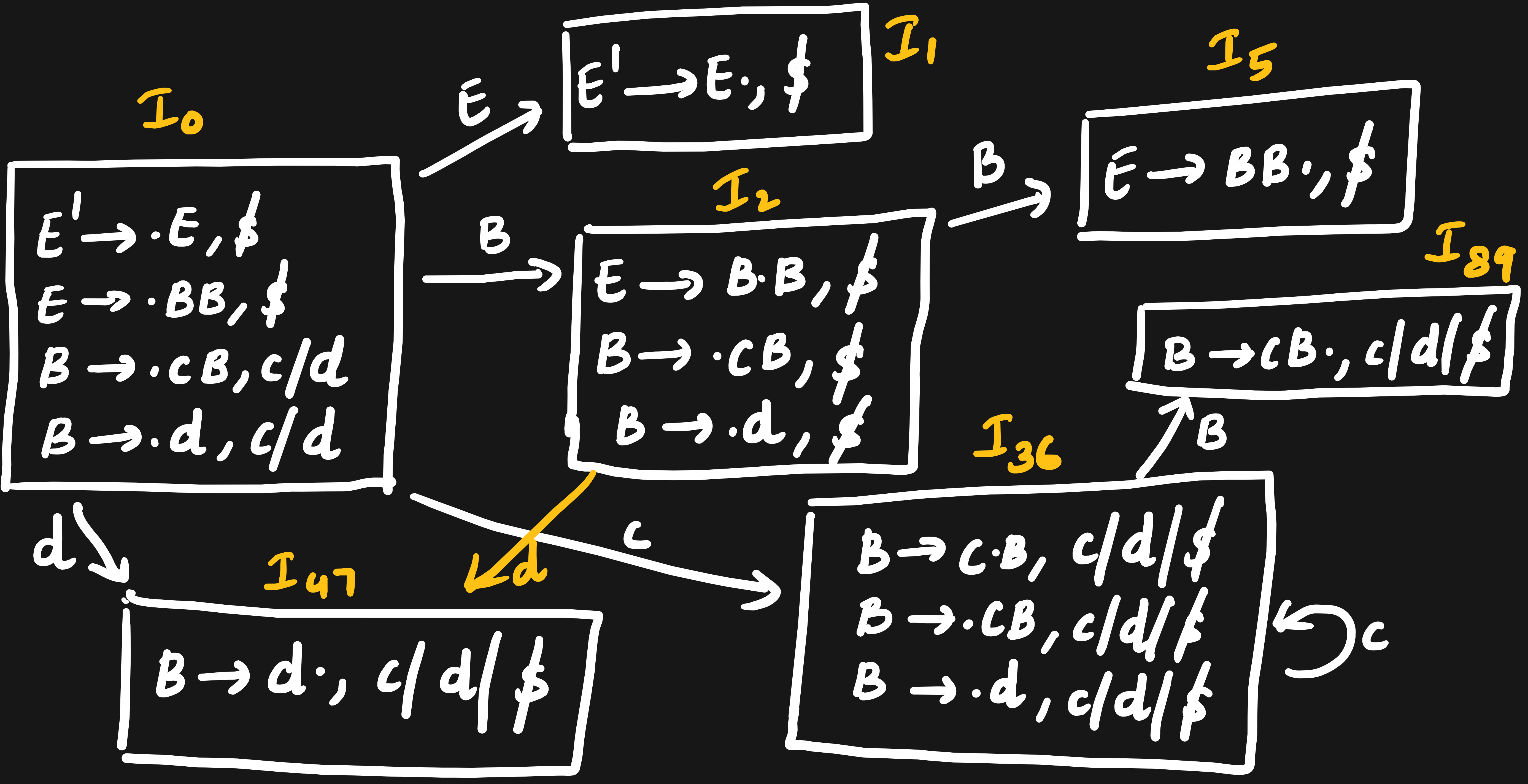
$B \rightarrow cB/d$



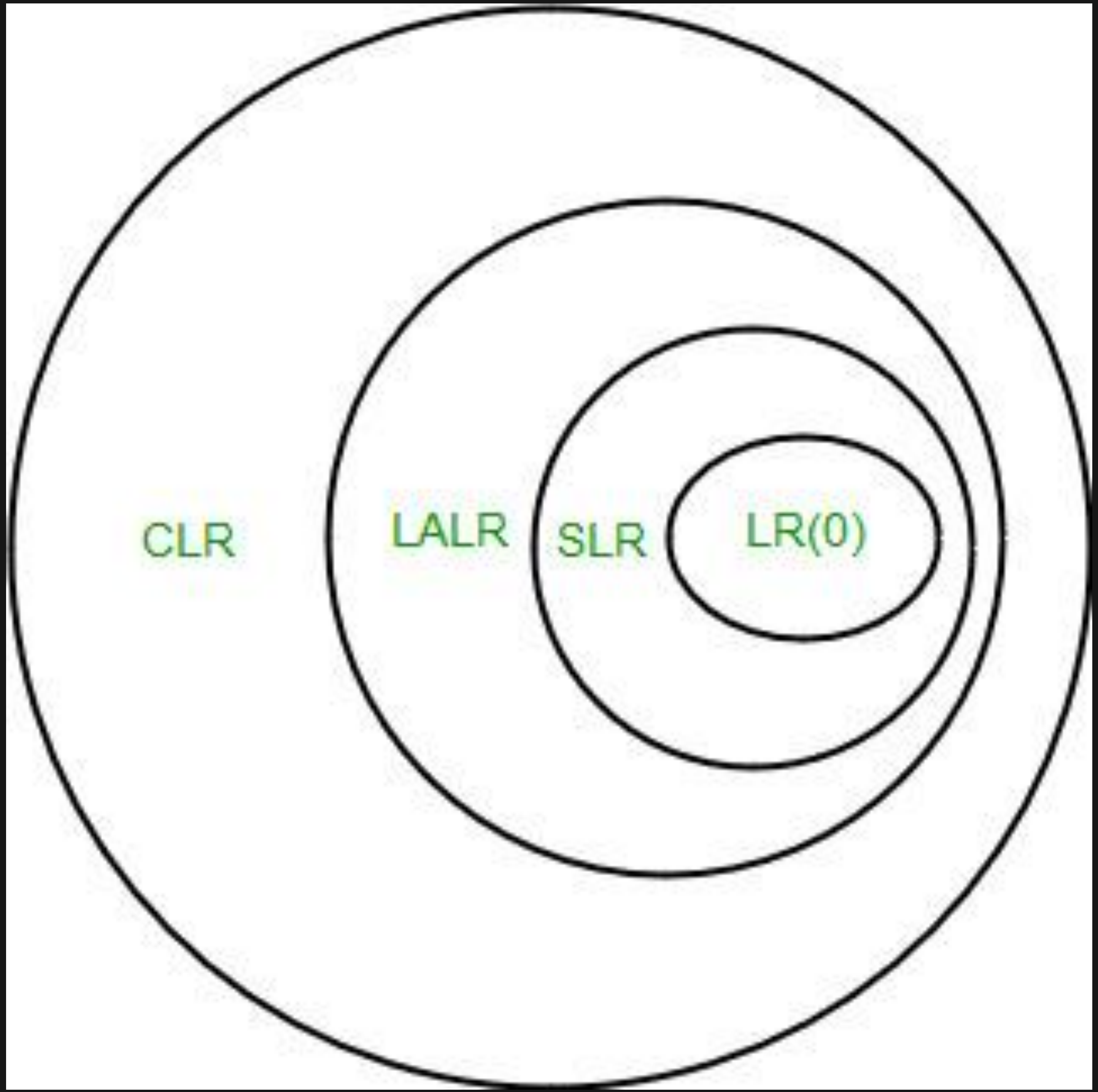
LALR PARSING TABLE

$I_3+I_6=I_{36}$
$I_4+I_7=I_{47}$
$I_8+I_9=I_{89}$

State	ACTION			GOTO	
	c	d	$	E	B
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		



Divys-Compiler Design PPT


CLR
LALR
SLR
LR(0)

SLR Parser	LALR Parser	CLR Parser
It is very easy and cheap to implement.	It is also easy and cheap to implement.	It is expensive and difficult to implement.
SLR Parser is the smallest in size.	LALR and SLR have the same size. As they have less number of states.	CLR Parser is the largest. As the number of states is very large.
Error detection is not immediate in SLR.	Error detection is not immediate in LALR.	Error detection can be done immediately in CLR Parser.
SLR fails to produce a parsing table for a certain class of grammars.	It is intermediate in power between SLR and CLR i.e., SLR ≤ LALR ≤ CLR.	It is very powerful and works on a large class of grammar.
It requires less time and space complexity.	It requires more time and space complexity.	It also requires more time and space complexity.