Syntax Analysis
2nd Phase of Compiler Construction

Section 3.1: LR(1)parsing

VALID ITEMS

- An item $A \rightarrow \beta_1 \bullet \beta_2$ is valid for a viable prefix $\alpha\beta_1$ if there is derivation $S' \overset{*}{\Rightarrow} \alpha A \omega \overset{*}{\Rightarrow} \alpha\beta_1\beta_2\omega$
- The fact that $A \rightarrow \beta_1 \bullet \beta_2$ is valid for $\alpha\beta_1$ tells us a lot about whether to shift/reduce when we find $\alpha\beta_1$ on the parsing stack.
- if β_2 is not an ε then, it suggests that we have not shifted the handle onto stack, so shift is our move
- if β_2 is ε then it looks as if $A \rightarrow \beta_1$ is the handle , and we should reduce by this production.

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.

LR(1) ITEM

- To avoid some of invalid reductions, the states need to carry more information.
- Extra information is put into a state by including a terminal symbol as a second component in an item.

- A LR(1) item is:

  $A \rightarrow \alpha \bullet \beta, a$ where **a** is the look-head of the LR(1) item

  (**a** is a terminal or \$.)

LR(1) ITEM (CONTI.)

- When β (in the LR(1) item $A \rightarrow \alpha \bullet \beta, a$) is not empty, the look- head does not have any affect.

- When β is empty ($A \rightarrow \alpha \bullet, a$), we do the reduction by $A \rightarrow \alpha$ only if the next input symbol is **a** (not for any terminal in FOLLOW(A)).

- A state will contain $A \rightarrow \alpha \bullet, a_1$ where $\{a_1,...,a_n\} \subseteq$ FOLLOW(A)

...

$A \rightarrow \alpha \bullet, a_n$

CANONICAL COLLECTION OF SETS OF LR(1) ITEMS

- The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, except that *closure* and *goto* operations work a little bit different.

- **closure(I)** is: (where I is a set of LR(1) items)
  - every LR(1) item in I is in closure(I)
  - if $A \rightarrow \alpha \bullet B\beta, a$ in closure(I) and $B \rightarrow \gamma$ is a production rule of G;

  then $B \rightarrow .\gamma, b$ will be in the closure(I) for each terminal b in FIRST(βa) .

GOTO OPERATION

- 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

- ***Algorithm***:

  C is { closure({S'→.S,$}) }

  repeat the followings until no more set of LR(1) items can be added to ***C***.

  for each 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***

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

A Short Notation for The Sets of LR(1) Items

- A set of LR(1) items containing the following items

$$A \rightarrow \alpha \bullet \beta, a_1$$

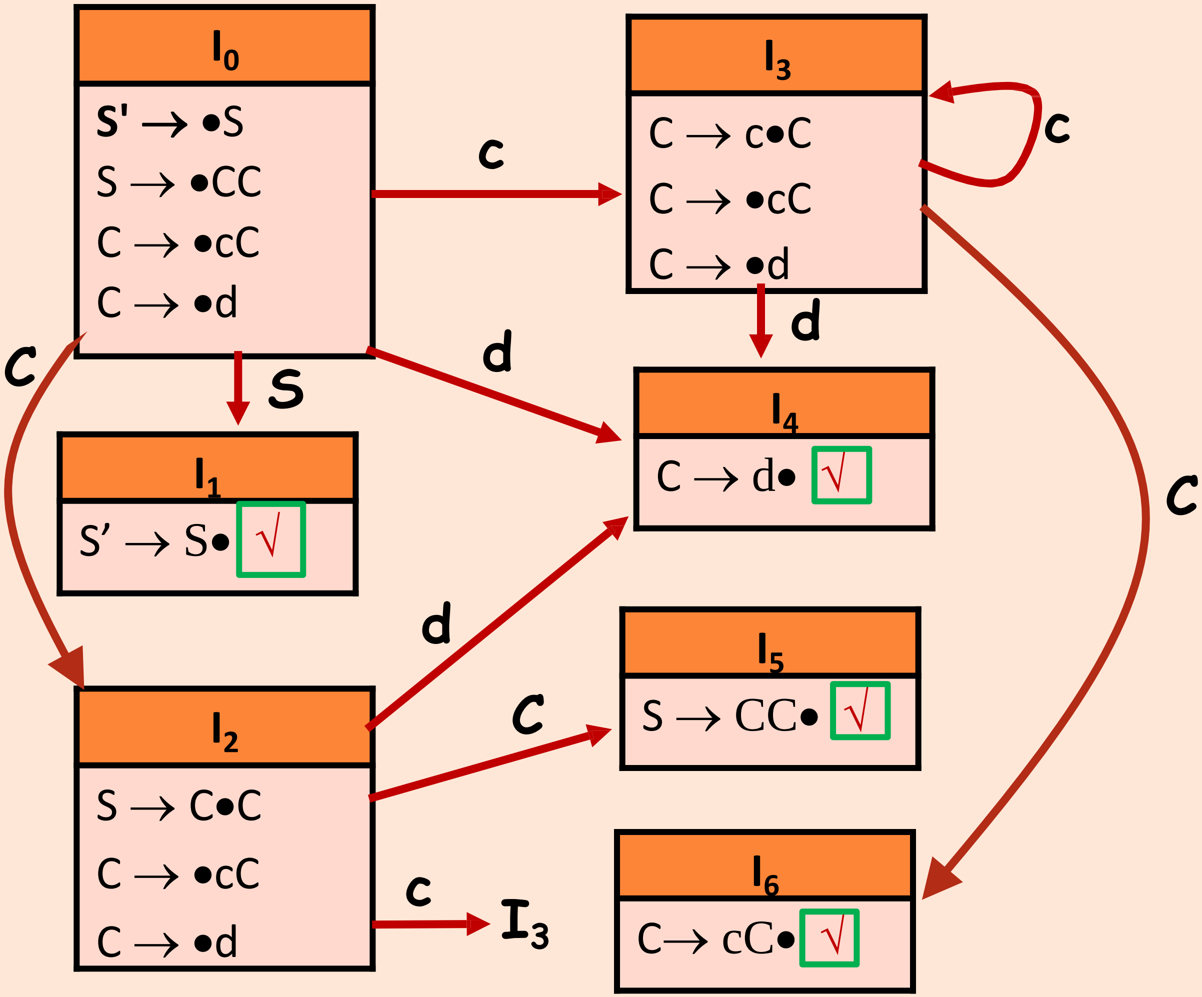
$$\ldots$$

$$A \rightarrow \alpha \bullet \beta, a_n$$

can be written as

$$A \rightarrow \alpha \bullet \beta, a_1/a_2/\ldots/a_n$$

THE CANONICAL LR(0) COLLECTION EX. 1



Initial Set

$S' \rightarrow \bullet S$

1) $S \rightarrow \bullet CC$
2) $C \rightarrow \bullet cC$
3) $C \rightarrow \bullet d$

THE CANONICAL LR(1) COLLECTION EX. 1

I_0

S' → •S, $
S → •CC
C → •cC
C → •d

Initial Set

S′ → •S
1) S → •CC
2) C → •cC
3) C → •d

S' → •S, $

A → α•Bβ,a

α is ε
B is S
β is ε
a is $
FIRST (β,a) is
FIRST(a) =$
All productions of S will have $ as look ahead

S → •CC

A→α•Bβ,a

α is ε
B is C
β is C
a is $
FIRST (β,a) is
FIRST(β) ={c, d}
All productions of C will have c and d as look ahead

C → •cC

No NT after •
No lookahead generation required

C → •d

No NT after •
No lookahead generation required

Final I_0
(With Lookaheads)

S' → •S, $
S → •CC, $
C → •cC , c/d
C → •d, c/d

The Canonical LR(1) Collection Ex. 1

I_0

S' → •S, $

S → •CC,$

C → •cC, c/d

C → •d, c/d

Shift • From left of ‘S’ to right of S

I_1

S′ → S•, $ √

THE CANONICAL LR(1) COLLECTION EX. 1

I_0

S' → •S, $
S → •CC,$
C → •cC, c/d
C → •d, c/d

Shift • From left of ‘C’ to right of ‘C’

I_1

S′ → S•, $ √

I_2

S → C•C, $
C → •cC
C → •d

S → C•C, $

A→ α•Bβ,a
α is C
B is C
β is ε
a is $
FIRST (β,a) is
FIRST(a) ={$}
All productions of C will have $ as look ahead

C → •cC, c/d

No NT after •
No lookahead generation required

C → •d, c/d

No NT after •
No lookahead generation required

Final I_2
(With Lookaheads)

S → C•C, $
C → •cC , $
C → •d, $

The Canonical LR(1) Collection Ex. 1

I_0

S' → •S, $
S → •CC,$
C → •cC, c/d
C → •d, c/d

Shift • From left of ‘c’ to right of ‘c’

I_1

S′ → S•, $ √

I_2

S → C•C, $
C → •cC, $
C → •d,$

I_3

C → c•C, c/d
C → •cC
C → •d

C → c•C, c/d

A→ α•Bβ,a
α is c
B is C
β is ε
a is {c,d}
FIRST (β,a) is
FIRST(a) ={c,d}
All productions of
C will have c/d as
look ahead

C → •cC

No NT after •
No lookahead
generation required

C → •d

No NT after •
No lookahead
generation required

Final I_3
(With Lookaheads)

S → c•C, c/d
C → •cC , c/d
C → •d, c/d

The Canonical LR(1) Collection Ex. 1

I_0

$S' \rightarrow \bullet S, \$$

$S \rightarrow \bullet CC, \$$

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

$C \rightarrow \bullet d, c/d$

I_3

$C \rightarrow c\bullet C, c/d$

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

$C \rightarrow \bullet d, c/d$

Shift • From left of 'd' to right of 'd'

I_4

$C \rightarrow d\bullet, c/d$ $\boxed{\surd}$

I_1

$S' \rightarrow S\bullet, \$$ $\boxed{\surd}$

I_2

$S \rightarrow C\bullet C, \$$

$C \rightarrow \bullet cC, \$$

$C \rightarrow \bullet d, \$$

THE CANONICAL LR(1) COLLECTION EX. 1

I_0

S' $\rightarrow \bullet S, \$$

$S \rightarrow \bullet CC,\$$

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

$C \rightarrow \bullet d, c/d$

I_1

$S' \rightarrow S\bullet, \$$ √

I_2

$S \rightarrow C\bullet C, \$$

$C \rightarrow \bullet cC, \$$

$C \rightarrow \bullet d,\$$

Shift • From left of 'C' to right of 'C'

I_3

$C \rightarrow c\bullet C, c/d$

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

$C \rightarrow \bullet d, c/d$

I_4

$C \rightarrow d\bullet, c/d$ √

I_5

$S \rightarrow CC\bullet, \$$ √

THE CANONICAL LR(1) COLLECTION EX. 1

I_0

S' → •S, $
S → •CC,$
C → •cC, c/d
C → •d, c/d

I_1

S′ → S•, $ √

I_2

S → C•C, $
C → •cC, $
C → •d,$

Shift • From left of ‘c’ to right of ‘c’

I_3

C → c•C, c/d
C → •cC, c/d
C → •d, c/d

I_4

C → d•, c/d √

I_5

S → CC•, $ √

I_6

C → c•C, $
C → •cC,
C → •d,

C → c•C,$

A → α•Bβ,a
α is c;
B is C
β is ε;
a is {$}
FIRST (β,a) is
FIRST(a) ={$}
All productions of C will have $ as look ahead

C → •cC

No NT after •
No lookahead generation required

C → •d

No NT after •
No lookahead generation required

Final I_6
(With Lookaheads)

C → c•C, $
C → •cC, $
C → •d, $

THE CANONICAL LR(1) COLLECTION EX. 1

I_0

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

I_1

$S' \rightarrow S\bullet, \$$ √

I_2

$S \rightarrow C\bullet C, \$$
$C \rightarrow \bullet cC, \$$
$C \rightarrow \bullet d,\$$

I_3

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

I_4

$C \rightarrow d\bullet, c/d$ √

I_5

$S \rightarrow CC\bullet, \$$ √

I_6

$C \rightarrow c\bullet C, \$$
$C \rightarrow \bullet cC, \$$
$C \rightarrow \bullet d, \$$

I_7

$C \rightarrow d\bullet, \$$ √

Shift • From left of ‘d’ to right of ‘d’

The Canonical LR(1) Collection Ex. 1

I_0

$S' \rightarrow \bullet S, \$$

$S \rightarrow \bullet CC, \$$

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

$C \rightarrow \bullet d, c/d$

I_1

$S' \rightarrow S\bullet, \$$ √

I_2

$S \rightarrow C\bullet C, \$$

$C \rightarrow \bullet cC, \$$

$C \rightarrow \bullet d, \$$

I_3

$C \rightarrow c\bullet C, c/d$

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

$C \rightarrow \bullet d, c/d$

I_4

$C \rightarrow d\bullet, c/d$ √

I_5

$S \rightarrow CC\bullet, \$$ √

I_6

$C \rightarrow c\bullet C, \$$

$C \rightarrow \bullet cC, \$$

$C \rightarrow \bullet d, \$$

I_7

$C \rightarrow d\bullet, \$$ √

Shift • From left of ‘C’ ‘to right of ‘C’

I_8

$C \rightarrow cC\bullet, c/d$ √

THE CANONICAL LR(1) COLLECTION EX. 1

I_0

S' → •S, \$
S → •CC,\$
C → •cC, c/d
C → •d, c/d

I_1

S′ → S•, \$ √

I_2

S → C•C, \$
C → •cC, \$
C → •d,\$

I_3

C → c•C, c/d
C → •cC, c/d
C → •d, c/d

Shift • From left of ‘c’
to right of ‘c’

I_4

C → d•, c/d √

I_5

S → CC•, \$ √

I_6

C → c•C, \$
C → •cC, \$
C → •d, \$

I_7

C → d•, \$ √

I_8

C → cC•, c/d √

THE CANONICAL LR(1) COLLECTION EX. 1

I_0

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

I_1

$S' \rightarrow S\bullet, \$$ √

I_2

$S \rightarrow C\bullet C, \$$
$C \rightarrow \bullet cC, \$$
$C \rightarrow \bullet d,\$$

I_3

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

Shift • From left of ‘d’
to right of ‘d’

I_4

$C \rightarrow d\bullet, c/d$ √

I_5

$S \rightarrow CC\bullet, \$$ √

I_6

$C \rightarrow c\bullet C, \$$
$C \rightarrow \bullet cC, \$$
$C \rightarrow \bullet d, \$$

I_7

$C \rightarrow d\bullet, \$$ √

I_8

$C \rightarrow cC\bullet, c/d$ √

THE CANONICAL LR(1) COLLECTION EX. 1

I_0

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

I_1

$S' \rightarrow S\bullet, \$$ √

I_2

$S \rightarrow C\bullet C, \$$
$C \rightarrow \bullet cC, \$$
$C \rightarrow \bullet d, \$$

I_3

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

I_4

$C \rightarrow d\bullet, c/d$ √

I_5

$S \rightarrow CC\bullet, \$$ √

I_6

$C \rightarrow c\bullet C, \$$
$C \rightarrow \bullet cC, \$$
$C \rightarrow \bullet d, \$$

I_7

$C \rightarrow d\bullet, \$$ √

I_8

$C \rightarrow cC\bullet, c/d$ √

I_9

$C \rightarrow cC\bullet, \$$ √

Shift • From left of ‘C’ to right of ‘C’

The Canonical LR(1) Collection Ex. 1

I_0

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

I_1

$S' \rightarrow S\bullet, \$$ √

I_2

$S \rightarrow C\bullet C, \$$
$C \rightarrow \bullet cC, \$$
$C \rightarrow \bullet d, \$$

I_3

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

I_4

$C \rightarrow d\bullet, c/d$ √

I_5

$S \rightarrow CC\bullet, \$$ √

I_6

$C \rightarrow c\bullet C, \$$
$C \rightarrow \bullet cC, \$$
$C \rightarrow \bullet d, \$$

Shift • From left of ‘c’ to right of ‘c’

I_7

$C \rightarrow d\bullet, \$$ √

I_8

$C \rightarrow cC\bullet, c/d$ √

I_9

$C \rightarrow cC\bullet, \$$ √

THE CANONICAL LR(1) COLLECTION EX. 1

I_0

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

I_1

$S' \rightarrow S\bullet, \$$ √

I_2

$S \rightarrow C\bullet C, \$$
$C \rightarrow \bullet cC, \$$
$C \rightarrow \bullet d, \$$

I_3

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

I_4

$C \rightarrow d\bullet, c/d$ √

I_5

$S \rightarrow CC\bullet, \$$ √

I_6

$C \rightarrow c\bullet C, \$$
$C \rightarrow \bullet cC, \$$
$C \rightarrow \bullet d, \$$

I_7

$C \rightarrow d\bullet, \$$ √

I_8

$C \rightarrow cC\bullet, c/d$ √

I_9

$C \rightarrow cC\bullet, \$$ √

Shift • From left of ‘d’
to right of ‘d’

The Canonical LR(1) Collection Ex. 1

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

I_1
$S' \to S\bullet, \$$ √

I_2
$S \to C\bullet C, \$$
$C \to \bullet cC, \$$
$C \to \bullet d, \$$

I_3
$C \to c\bullet C, c/d$
$C \to \bullet cC, c/d$
$C \to \bullet d, c/d$

I_4
$C \to d\bullet, c/d$ √

I_5
$S \to CC\bullet, \$$ √

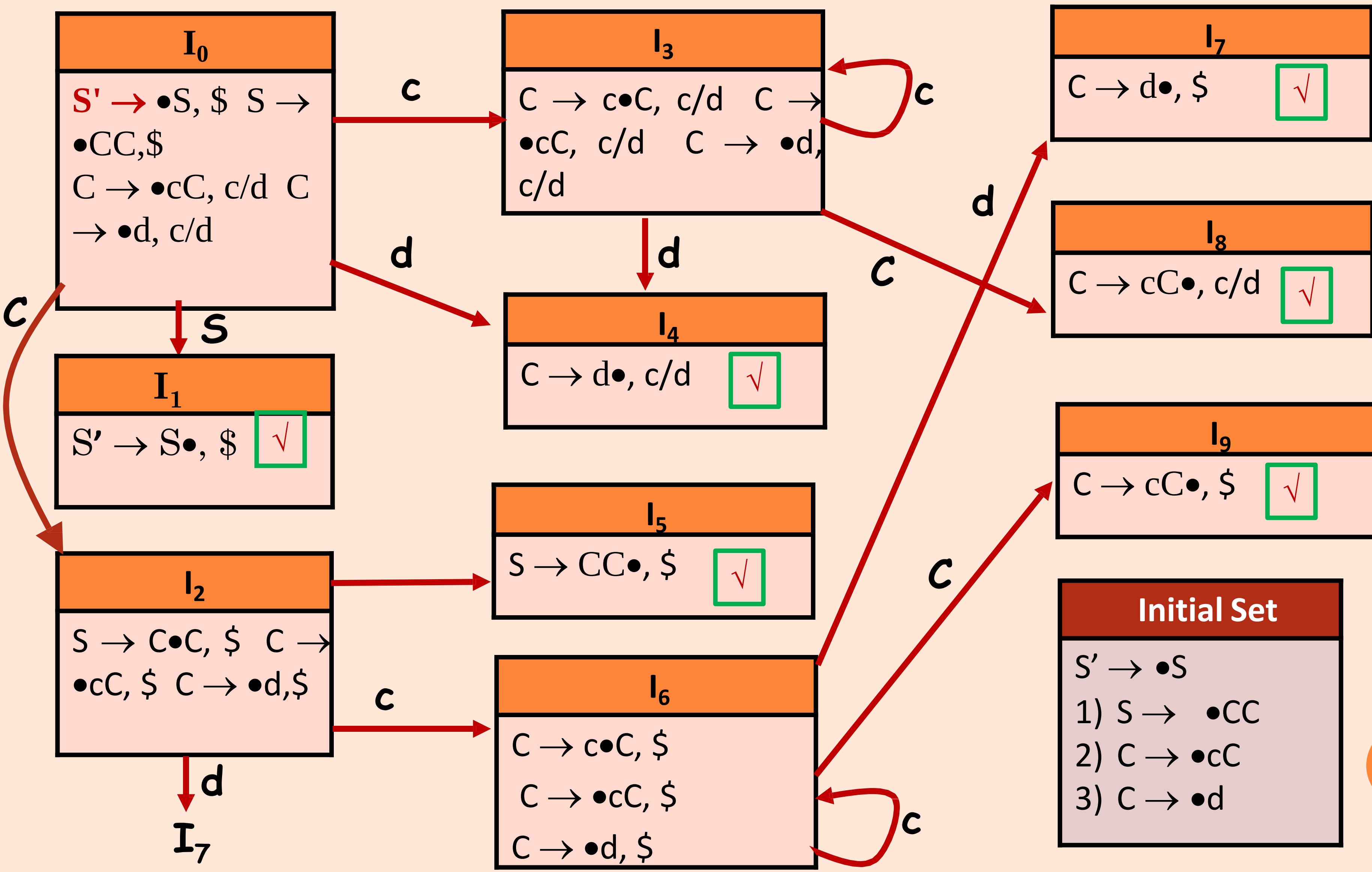
I_6
$C \to c\bullet C, \$$
$C \to \bullet cC, \$$
$C \to \bullet d, \$$

I_7
$C \to d\bullet, \$$ √

I_8
$C \to cC\bullet, c/d$ √

I_9
$C \to cC\bullet, \$$ √

THE CANONICAL LR(1) COLLECTION EX. 1



THE CANONICAL LR(0) AND LR(1)

Initial Set

S′ → •S
1) S → •CC
2) C → •cC
3) C → •d

I_0

S' → •S
S → •CC
C → •cC
C → •d

I_1

S′ → S• √

I_2

S → C•C
C → •cC
C → •d

I_3

C → c•C
C → •cC
C → •d

I_4

C → d• √

I_5

S → CC• √

I_6

C → cC• √

I_0

S' → •S, $
S → •CC,$
C → •cC, c/d
C → •d, c/d

I_1

S′ → S•, $ √

I_2

S → C•C, $
C → •cC, $
C → •d,$

I_3

C → c•C, c/d
C → •cC, c/d
C → •d, c/d

I_4

C → d•, c/d √

I_5

S → CC•, $ √

I_6

C → c•C, $
C → •cC, $
C → •d, $

I_7

C → d•, $ √

I_8

C → cC•, c/d √

I_9

C → cC•, $ √

CONSTRUCTION OF LR(1) PARSING TABLES

1. Construct the canonical collection of sets of LR(1) items for G'.
   $C \leftarrow \{I_0,...,I_n\}$

2. Create the parsing action table as follows
   - If a is a terminal, $A \rightarrow \alpha \bullet a\beta,b$ in I_i and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j.***
   - If $A \rightarrow \alpha \bullet ,a$ is in I_i , then action[i,a] is ***reduce*** $\boldsymbol{A \rightarrow \alpha}$ where $A \neq S'$.
   - If $S' \rightarrow S \bullet ,\$$ is in I_i , then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not LR(1).

3. Create the parsing goto table
   - for all non-terminals A, 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 contains $S' \rightarrow .S,\$$

LR(1) PARSING TABLES – EX. 1

I_0

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

I_1

$S' \to S\bullet, \$$ √

I_2

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

I_3

$C \to c\bullet C, c/d$
$C \to \bullet cC, c/d$
$C \to \bullet d, c/d$

I_4

$C \to d\bullet, c/d$ √

I_5

$S \to CC\bullet, \$$ √

I_6

$C \to c\bullet C, \$$
$C \to \bullet cC, \$$
$C \to \bullet d, \$$

I_7

$C \to d\bullet, \$$ √

I_8

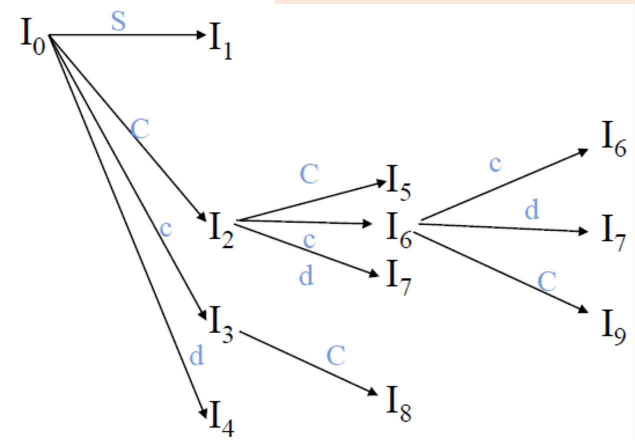
$C \to cC\bullet, c/d$ √

I_9

$C \to cC\bullet, \$$ √

Initial Set

$S' \to \bullet S$
1) $S \to \bullet CC$
2) $C \to \bullet cC$
3) $C \to \bullet d$



	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		

•S*i* means shift and stack state *i*
•r*j* means reduce by production numbered *j*
acc means accept state
•blank mean error

ACTIONS OF A (S)LR-PARSER -- EXAMPLE

Stack	Input	Action	Output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3 (GOTO)	*id+id$	reduce by T→F	T→F
0T2(GOTO)	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10 (GOTO)	+id$	reduce by T→T*F	T→T*F
0T2 (GOTO)	+id$	reduce by E→T	E→T
0E1(GOTO)	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3 (GOTO)	$	reduce by T→F	T→F
0E1+6T9 (GOTO)	$	reduce by E→E+T	E→E+T
0E1	$	accept	

	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		

E' → .E
1) **E → E+T**
2) **E → T**
3) **T → T*F**
4) **T → F**
5) **F → (E)**
6) **F → id**

LR(1) PARSING TABLES –EX.2

State	ACTION				GOTO		
	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			
10				r5			
11	s12	s11				10	13
12				r4			
13				r3			

Initial Grammar

S' → •S
1)S → •L=R
2)S → •R
3) L → •*R
4)L → •id
5)R → •L

•S*i* means shift and stack state *i*
•r*j* means reduce by production numbered *j*
acc means accept state
•blank mean error

Section 3.2: Look Ahead LR (LALR)

LALR PARSING TABLES

- **LALR** stands for **LookAhead 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.
- But 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.

THE CANONICAL LR(1) COLLECTION EX. 1

Initial Set

$S' \to \bullet S$

1) $S \to \bullet CC$

2) $C \to \bullet cC$

3) $C \to \bullet d$

I_0

$S' \to \bullet S, \$$

$S \to \bullet CC, \$$

$C \to \bullet cC, c/d$

$C \to \bullet d, c/d$

I_1

$S' \to S\bullet, \$$ √

I_2

$S \to C\bullet C, \$$

$C \to \bullet cC, \$$

$C \to \bullet d, \$$

I_3

$C \to c\bullet C, c/d$

$C \to \bullet cC, c/d$

$C \to \bullet d, c/d$

I_4

$C \to d\bullet, c/d$ √

I_5

$S \to CC\bullet, \$$ √

I_6

$C \to c\bullet C, \$$

$C \to \bullet cC, \$$

$C \to \bullet d, \$$

I_7

$C \to d\bullet, \$$ √

I_8

$C \to cC\bullet, c/d$ √

I_9

$C \to cC\bullet, \$$ √

I_3 and I_6

I_4 and I_7

I_8 and I_9

THE CANONICAL LR(1) COLLECTION EX. 2

Initial Grammar

S' → •S
1) S → •L=R
2) S → •R
3) L → •*R
4) L → •id
5) R → •L

I_0

S' → •S,$
S → •L=R, $
S → •R, $
L → •*R,=/$
L → •id,=/$
R → •L ,$

I_1

S' → S•,$ √

I_2

S → L•=R,$
R → L• ,$

I_3

S→ R•, $ √

I_4

L → *•R,=/$
R → •L,=/ $
L → •*R, =/$
L → •id,=/$

I_5

L →id• =/$ √

I_6

S → L=•R,$
R → •L, $
L → •*R,$
L → •id,$

I_7

L → *R•, =/$ √

I_8

R → L•, =/$ √

I_9

S → L=R•, $ √

I_{10}

R → L•, $ √

I_{11}

L → *•R,$
R → •L, $
L → •*R,$
L → •id,$

I_{12}

L →id•, $ √

I_{13}

L → *R•, $ √

I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

CREATING LALR PARSING TABLES

Canonical LR(1) Parser ➔ LALR Parser

shrink # of states

- This shrink process may introduce a **reduce/reduce** conflict in the resulting LALR parser (so the grammar is NOT LALR)
- But, this shrink process does not produce a **shift/reduce** conflict.

THE CORE OF A SET OF LR(1) ITEMS

- The core of a set of LR(1) items is the set of its first component.

Ex: $S \rightarrow L \bullet = R, \$$ ➔ $S \rightarrow L \bullet = R$ Core

$R \rightarrow L \bullet, \$$ $R \rightarrow L \bullet$

- We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

$I_1 : L \rightarrow id \bullet, =$ A new state: $I_{12} : L \rightarrow id \bullet, =$

➔ $L \rightarrow id \bullet, \$$

$I_2 : L \rightarrow id \bullet, \$$ have same core, merge them

- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

Creation of LALR Parsing Tables

- Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
- Find each core; find all sets having that same core; replace those sets having same cores with a single set which is their union.

  $C=\{I_0,...,I_n\} \Rightarrow C'=\{J_1,...,J_m\}$ where $m \leq n$
- Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
  - Note that: If $J=I_1 \cup ... \cup I_k$ since $I_1,...,I_k$ have same cores

    $\Rightarrow$ cores of $goto(I_1,X),...,goto(I_2,X)$ must be same.
  - So, $goto(J,X)=K$ where K is the union of all sets of items having same cores as $goto(I_1,X)$.
- If no conflict is introduced, the grammar is LALR(1) grammar. (We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)

The Canonical LR(1) Collection Ex. 1

Initial Set

S' → •S
1) S → •CC
2) C → •cC
3) C → •d

I_0

S' → •S, $
S → •CC,$
C → •cC, c/d
C → •d, c/d

I_1

S' → S•, $ √

I_2

S → C•C, $
C → •cC, $
C → •d,$

I_3

C → c•C, c/d
C → •cC, c/d
C → •d, c/d

I_4

C → d•, c/d √

I_5

S → CC•, $ √

I_6

C → c•C, $
C → •cC, $
C → •d, $

I_7

C → d•, $ √

I_8

C → cC•, c/d √

I_9

C → cC•, $ √

Same Core

I_3 and I_6

I_4 and I_7

I_8 and I_9

The Canonical LR(1) Collection Ex. 1

I_0

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

I_1

$S' \rightarrow S\bullet, \$$ √

I_2

$S \rightarrow C\bullet C, \$$
$C \rightarrow \bullet cC, \$$
$C \rightarrow \bullet d, \$$

I_{36}

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

I_{47}

$C \rightarrow d\bullet, c/d/\$$ √

I_5

$S \rightarrow CC\bullet, \$$ √

I_{89}

$C \rightarrow cC\bullet, c/d/\$$ √

Initial Set

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

Same Core

I_3 and I_6

I_4 and I_7

I_8 and I_9

LALR(1) Parsing Tables –Ex. 1

	ACTION				GOTO	
State	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			

The Canonical LR(1) Collection Ex. 2

Initial Grammar

S' → •S
1) S → •L=R
2) S → •R
3) L → •*R
4) L → •id
5) R → •L

I_0

S' → •S,$
S → •L=R, $
S → •R, $
L → •*R,=/$
L → •id,=/$
R → •L ,$

I_1

S' → S•,$ √

I_2

S → L•=R,$
R → L• ,$

I_3

S→ R•, $ √

I_4

L → *•R,=/$
R → •L,=/ $
L → •*R, =/$
L → •id,=/$

I_5

L →id• =/$ √

I_6

S → L=•R,$
R → •L, $
L → •*R,$
L → •id,$

I_7

L → *R•, =/$ √

I_8

R → L•, =/$ √

I_9

S → L=R•, $ √

I_{10}

R → L•, $ √

I_{11}

L → *•R,$
R → •L, $
L → •*R,$
L → •id,$

I_{12}

L →id•, $ √

I_{13}

L → *R•, $ √

Same Core

I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

The Canonical LR(1) Collection Ex. 2

I_0

S' → •S,$
S → •L=R, $
S → •R, $
L → •*R,=/$
L → •id,=/$
R → •L ,$

I_1

S' → S•,$ √

I_2

S → L•=R,$
R → L• ,$

I_3

S→ R•, $ √

I_{411}

L → *•R,=/$
R → •L,=/ $
L → •*R, =/$
L → •id,=/$

I_{512}

L →id• =/$ √

I_6

S → L=•R,$
R → •L, $
L → •*R,$
L → •id,$

I_{713}

L → *R•, =/$ √

I_{810}

R → L•, =/$ √

I_9

S → L=R•, $ √

Initial Grammar

S' → •S
1) S → •L=R
2) S → •R
3) L → •*R
4) L → •id
5) R → •L

Same Core

I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

LALR(1) PARSING TABLES –EX. 2

	ACTION				GOTO		
State	**id**	*****	**=**	**$**	**S**	**L**	**R**
0	s_{512}	s_{411}			1	2	3
1				acc			
2			s_6	r_5			
3				r_2			
4	s_{512}	s_{411}				810	713
5			r_4	r_4			
6	s_{512}	s_{11}				810	9
7			r_3	r_3			
8			r_5	r_5			
9				r_1			

Shift/Reduce Conflict

- We say that we cannot introduce a shift/reduce conflict during the shrink process for the creation of the states of a LALR parser.
- Assume that we can introduce a shift/reduce conflict. In this case, a state of LALR parser must have:

$$A \rightarrow \alpha\bullet,a \qquad \text{and} \qquad B \rightarrow \beta\bullet a\gamma,b$$

- This means that a state of the canonical LR(1) parser must have:

$$A \rightarrow \alpha\bullet,a \qquad \text{and} \qquad B \rightarrow \beta\bullet a\gamma,c$$

But, this state has also a shift/reduce conflict. i.e. The original canonical LR(1) parser has a conflict.

(Reason for this, the shift operation does not depend on lookaheads)

REDUCE/REDUCE CONFLICT

- But, we may introduce a reduce/reduce conflict during the shrink process for the creation of the states of a LALR parser.

for acd, ace, bcd, bce, LR(0) items are

Initial Grammar
$S' \to S$
$S \to aAd \mid bBd \mid aBe \mid bAe$
$A \to c$
$B \to c$

I_1
$A \to c\bullet$,d
$B \to c\bullet$,e

I_2
$A \to c\bullet$, e
$B \to c\bullet$,d

Reduce/Reduce Conflict ⟹

I_{12}
$A \to c\bullet$, e/d
$B \to c\bullet$,e/d

LALR(1) PARSING TABLES –EX. 2

	ACTION					GOTO		
State	id	*	=	$		S	L	R
0	s512	s411				1	2	3
1				acc				
2			s6	r5				
3				r2				
4	s512	s411					810	713
5			r4	r4				
6	s512	s11					810	9
7			r3	r3				
8			r5	r5				
9				r1				

No shift/reduce
No reduce/reduce conflict

$\Downarrow$

So, it is a LALR(1) grammar

Section 3.3: Error and Error Recovery

ERRORS

- **Lexical errors** include misspellings of identifiers, keywords, or operators -e.g., the use of an identifier elipsesize instead of ellipsesize - and missing quotes around text intended as a string.

- **Syntactic errors** include misplaced semicolons or extra or missing braces; that is, "{" or "}". As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error (however, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code).

ERRORS –CONTD.

- **Semantic errors** include type mismatches between operators and operands. An example is a return statement in a Java method with result type void.

- **Logical errors** can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==. The program containing = may be well formed; however, it may not reflect the programmer's intent.

CHALLENGES OF ERROR HANDLER

- The error handler in a parser has goals that are simple to state but challenging to realize:
  - Report the presence of errors clearly and accurately.
  - Recover from each error quickly enough to detect subsequent errors.
  - Add minimal overhead to the processing of correct programs.

ERROR RECOVERY TECHNIQUES

- Panic-Mode Error Recovery
  - Skipping the input symbols until a synchronizing token is found.
- Phrase-Level Error Recovery
  - Each empty entry in the parsing table is filled with a pointer to a specific error routine to take care that error case.
- Error-Productions
  - If we have a good idea of the common errors that might be encountered, we can augment the grammar with productions that generate erroneous constructs.
  - When an error production is used by the parser, we can generate appropriate error diagnostics.
  - Since it is almost impossible to know all the errors that can be made by the programmers, this method is not practical.
- Global-Correction
  - Ideally, we would like a compiler to make as few change as possible in processing incorrect inputs.
  - We have to globally analyze the input to find the error.
  - This is an expensive method, and it is not in practice.

53

Error Recovery in Predictive Parsing

- An error may occur in the predictive parsing (LL(1) parsing)
  - if the terminal symbol on the top of stack does not match with the current input symbol.
  - if the top of stack is a non-terminal A, the current input symbol is a, and the parsing table entry M[A,a] is empty.
- What should the parser do in an error case?
  - The parser should be able to give an error message (as much as possible meaningful error message).
  - It should recover from that error case, and it should be able to continue the parsing with the rest of the input.

54

Panic-Mode Error Recovery in LL(1) Parsing

- In panic-mode error recovery, we skip all the input symbols until a synchronizing token is found.
- What is the synchronizing token?
  - All the terminal-symbols in the follow set of a non-terminal can be used as a synchronizing token set for that non-terminal.
- So, a simple panic-mode error recovery for the LL(1) parsing:
  - All the empty entries are marked as ***synch*** to indicate that the parser will skip all the input symbols until a symbol in the follow set of the non-terminal A which on the top of the stack. Then the parser will pop that non-terminal A from the stack. The parsing continues from that state.
  - To handle unmatched terminal symbols, the parser pops that unmatched terminal symbol from the stack and it issues an error message saying that that unmatched terminal is inserted.

55

Panic-Mode Error Recovery - Example

$S \to AbS \mid e \mid \varepsilon$

$A \to a \mid cAd$

FOLLOW(S)={$}

FOLLOW(A)={b,d}

	a	b	c	d	e	$
S	$S \to AbS$	***sync***	$S \to AbS$	***sync***	$S \to e$	$S \to \varepsilon$
A	$A \to a$	***sync***	$A \to cAd$	***sync***	***sync***	***sync***

stack	input	output
$S	aab$	$S \to AbS$
$SbA	aab$	$A \to a$
$Sba	aab$	
$Sb	ab$	Error: missing b, inserted
$S	ab$	$S \to AbS$
$SbA	ab$	$A \to a$
$Sba	ab$	
$Sb	b$	
$S	$	$S \to \varepsilon$
$	$	accept

stack	input	output
$S	ceadb$	$S \to AbS$
$SbA	ceadb$	$A \to cAd$
$SbdAc	ceadb$	
$SbdA	eadb$	Error: unexpected e (illegal A)
(Remove all input tokens until first b or d, pop A)		
$Sbd	db$	
$Sb	b$	
$S	$	$S \to \varepsilon$
$	$	accept

PHRASE-LEVEL ERROR RECOVERY

- Each empty entry in the parsing table is filled with a pointer to a special error routine which will take care that error case.
- These error routines may:
  - change, insert, or delete input symbols.
  - issue appropriate error messages
  - pop items from the stack.
- We should be careful when we design these error routines, because we may put the parser into an infinite loop.

Error Recovery in LR Parsing

- An LR parser will detect an error when it consults the parsing action table and finds an error entry. All empty entries in the action table are error entries.
- Errors are never detected by consulting the goto table.
- An LR parser will announce error as soon as there is no valid continuation for the scanned portion of the input.
- A canonical LR parser (LR(1) parser) will never make even a single reduction before announcing an error.
- The SLR and LALR parsers may make several reductions before announcing an error.
- But, all LR parsers (LR(1), LALR and SLR parsers) will never shift an erroneous input symbol onto the stack.

PANIC MODE ERROR RECOVERY IN LR PARSING

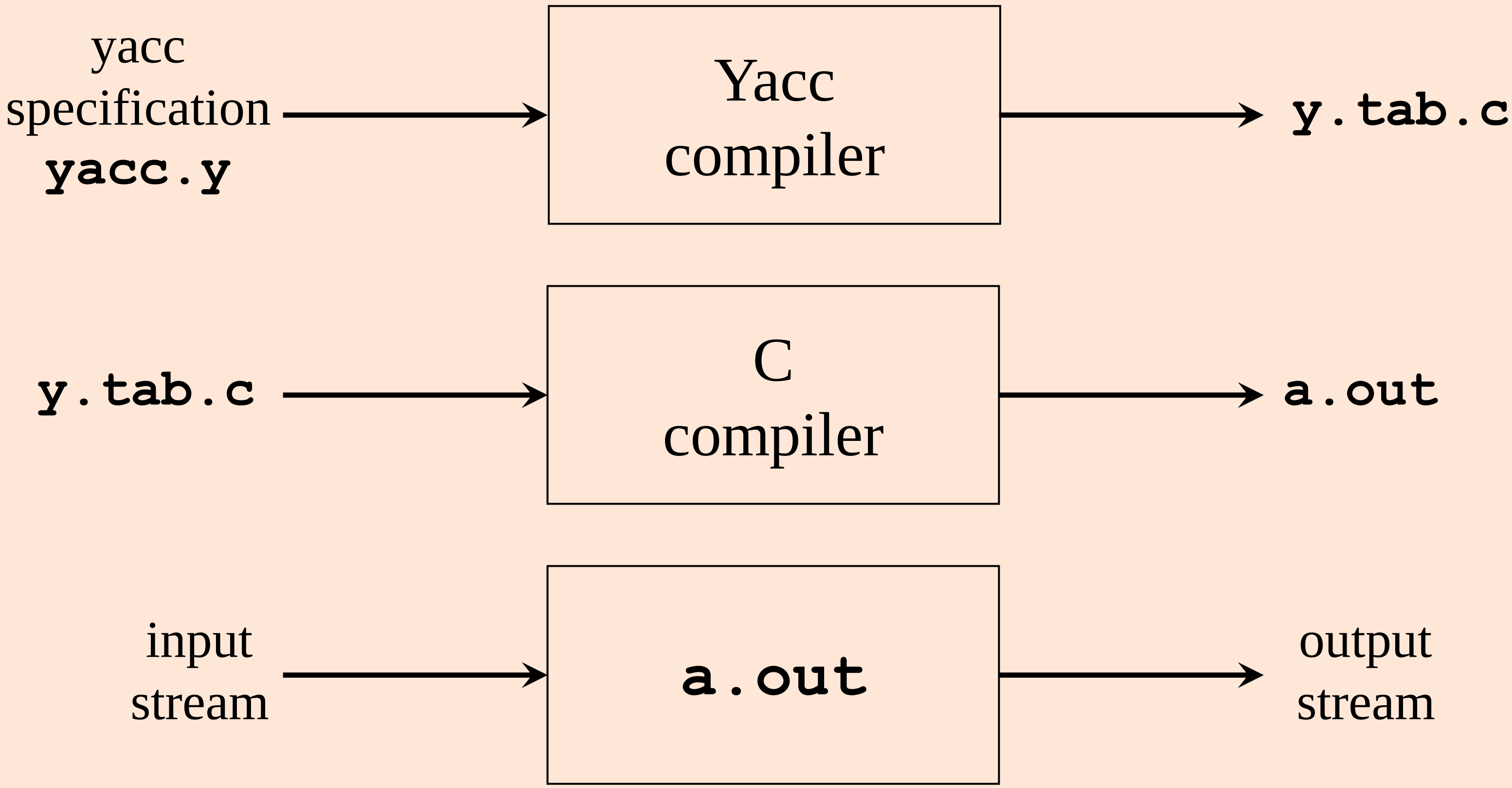
- Scan down the stack until a state **s** with a goto on a particular nonterminal **A** is found. (Get rid of everything from the stack before this state s).
- Discard zero or more input symbols until a symbol **a** is found that can legitimately follow A.
  - The symbol a is simply in FOLLOW(A), but this may not work for all situations.
- The parser stacks the nonterminal **A** and the state **goto[s,A]**, and it resumes the normal parsing.
- This nonterminal A is normally is a basic programming block (there can be more than one choice for A).
  - stmt, expr, block, ...

Phrase-Level Error Recovery in LR Parsing

- Each empty entry in the action table is marked with a specific error routine.
- An error routine reflects the error that the user most likely will make in that case.
- An error routine inserts the symbols into the stack or the input (or it deletes the symbols from the stack and the input, or it can do both insertion and deletion).
  - missing operand
  - unbalanced right parenthesis

Section 3.4: Yet Another Compiler Compiler (YACC)

Creating an LALR(1) Parser with YACC



YACC SPECIFICATIONS

- A *yacc specification* consists of three parts:

  yacc declarations, and C declarations within `%{ %}`
  `%%`
  translation rules
  `%%`
  user-defined auxiliary procedures

- The *translation rules* are productions with actions:

  $production_1$ { $semantic\ action_1$ }
  $production_2$ { $semantic\ action_2$ }
  …
  $production_n$ { $semantic\ action_n$ }

Writing a Grammar in Yacc

- Productions in Yacc are of the form

```
Nonterminal  : tokens/nonterminals { action }
             | tokens/nonterminals { action }
             …
             ;
```

- Tokens that are single characters can be used directly within productions, e.g. `'+'`
- Named tokens must be declared first in the declaration part using

  `%token` *TokenName*

Example 1

```
%{ #include <ctype.h> %}
%token DIGIT
%%
line    : expr '\n'           { printf("%d\n", $1); }
        ;
expr    : expr '+' term       { $$ = $1 + $3; }
        | term                { $$ = $1; }
        ;
term    : term '*' factor     { $$ = $1 * $3; }
        | factor              { $$ = $1; }
        ;
factor  : '(' expr ')'        { $$ = $2; }
        | DIGIT               { $$ = $1; }
        ;
%%
int yylex()
{ int c = getchar();
  if (isdigit(c))
  { yylval = c-'0';
    return DIGIT;
  }
  return c;
}
```

Also results in definition of
`#define DIGIT xxx`

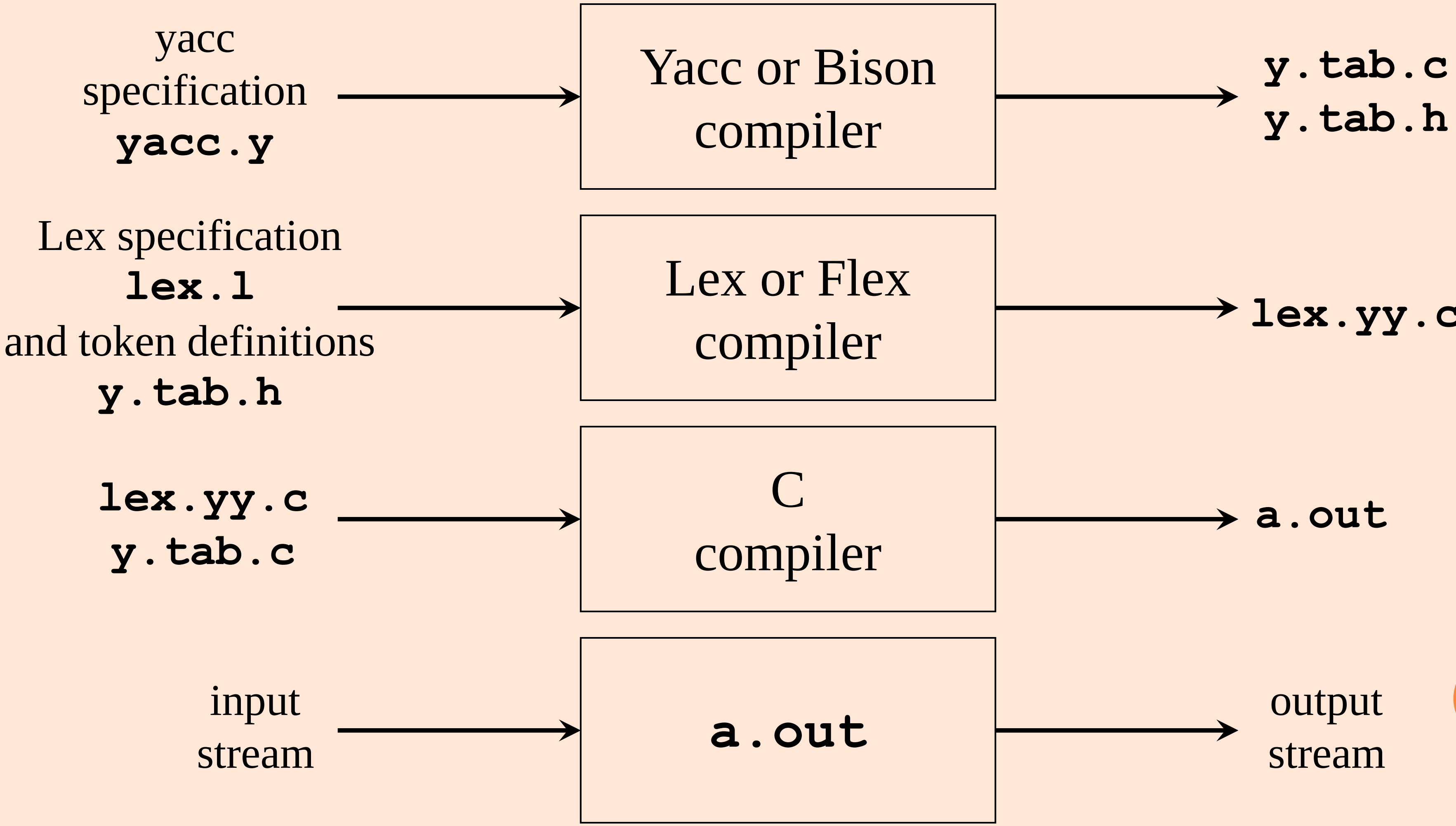
Attribute of **`term`** (parent)

Attribute of **`factor`** (child)

Attribute of token (stored in **`yylval`**)

Example of a very crude lexical analyzer invoked by the parser

Combining Lex/Flex with Yacc/Bison



ERROR RECOVERY IN YACC

```
%{
…
%}
…
%%
lines   : lines expr '\n'       { printf("%g\n", $2; }
        | lines '\n'
        | /* empty */
        | error '\n'            { yyerror("reenter last line: ");
                                  yyerrok;
                                }
        ;
…
```

Error production:
set error mode and
skip input until newline

Reset parser to normal mode