Syntax Analysis

Introduction

- Every language has rules that prescribe the syntactic structure of well formed programs.

- The syntax can be described using Context Free Grammars (CFG) or Backus-Naur Form (BNF) notation.

Introduction (cont'd)

- The use of CFG has several advantages over BNF:
  - helps in identifying ambiguities
  - a grammar gives a precise yet easy to understand syntactic specification of a programming language
  - it is possible to have a tool which automatically produces a parser using the grammar
  - a properly designed grammar helps in modifying the parser easily when the language changes

Syntax error handling

- The error handler should be written with the following goals in mind:
  - errors should be reported clearly and accurately
  - the compiler should recover and detect other errors
  - it should not slow down the whole process (of compilation) significantly
- Fortunately most errors are straight forward and easy to detect.

  E.g. of common errors:

  missing ; missing : in :=

Syntax error handling (cont'd)

- The error handler should report:
  - the place of the error
  - the type of the error (if possible)

- There are four main strategies in error handling:
  - **Panic mode**: discards all tokens until a synchronization token is found.
  - **Phrase level recovery**: the parser makes a local correction so that it can continue to parse the rest of the input.

Syntax error handling (cont'd)

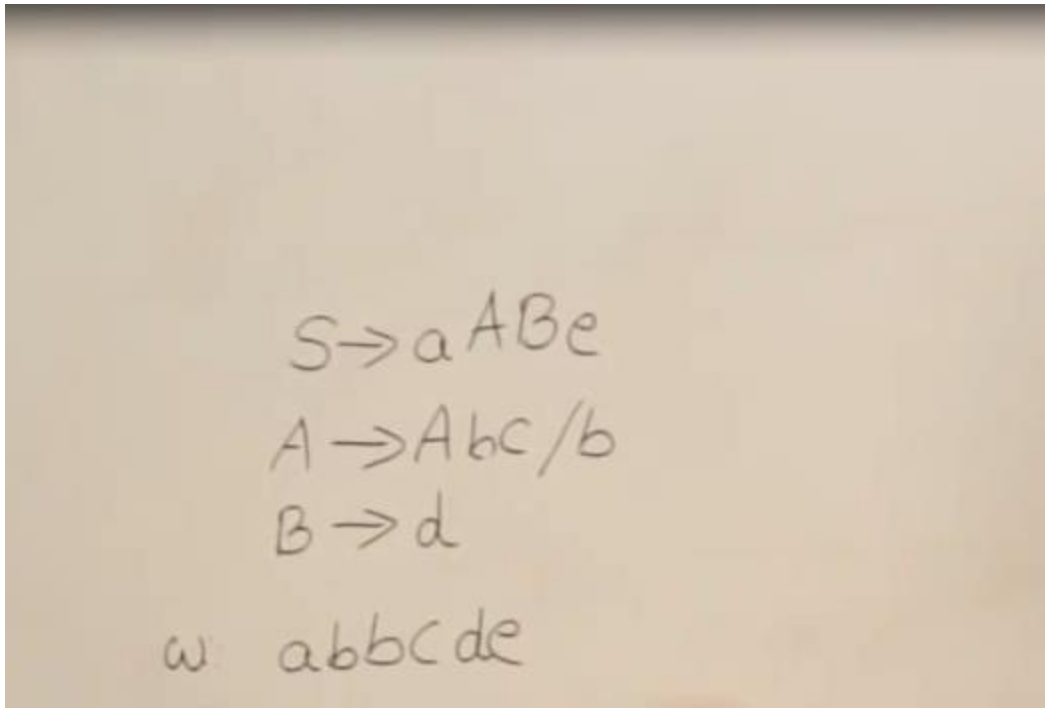
- **Error productions**: augment the grammar to capture the most common errors that programmers make.
- **Global correction**: makes as few changes as possible in the program so that a globally least cost correction program is obtained.

- Read more: textbook

Parsing techniques

- Two general approaches to parsing:
  - Top-down parsing
    - Recursive Descent Parsing
    - Non-recursive Predictive Parsing
  - Bottom-up parsing
    - LR Parsing

example



Recursive Descent Parsing (RDP)

- This method of top-down parsing can be considered as an attempt to find the left most derivation for an input string .
- It may involve **backtracking** (make repeated scans of the input)

  ➔ inefficient

RDP (cont'd)

- To construct the parse tree using RDP:
  - create a one node tree consisting of S
  - two pointers, one for the tree and one for the input, will be used to indicate where the parsing process is
  - initially, they will be on S and the first input symbol, respectively
  - then we use the first S-production to expand the tree. The tree pointer will be positioned on the left most symbol of the newly created sub-tree.

RDP (cont'd)

- as the symbol pointed by the tree pointer matches that of the symbol pointed by the input pointer, both pointers are moved to the right
- whenever the tree pointer points on a non-terminal, we expand it using the first production of the non-terminal.
- whenever the pointers point on different terminals, the production that was used is not correct, thus another production should be used. We have to go back to the step just before we replaced the non-terminal and use another production.
- if we reach the end of the input and the tree pointer passes the last symbol of the tree, we have finished parsing.

RDP (cont'd)

- Exercise:

  G: S → cAd

  A → ab|a

Draw the parse tree for the input string **cad** using RDP.

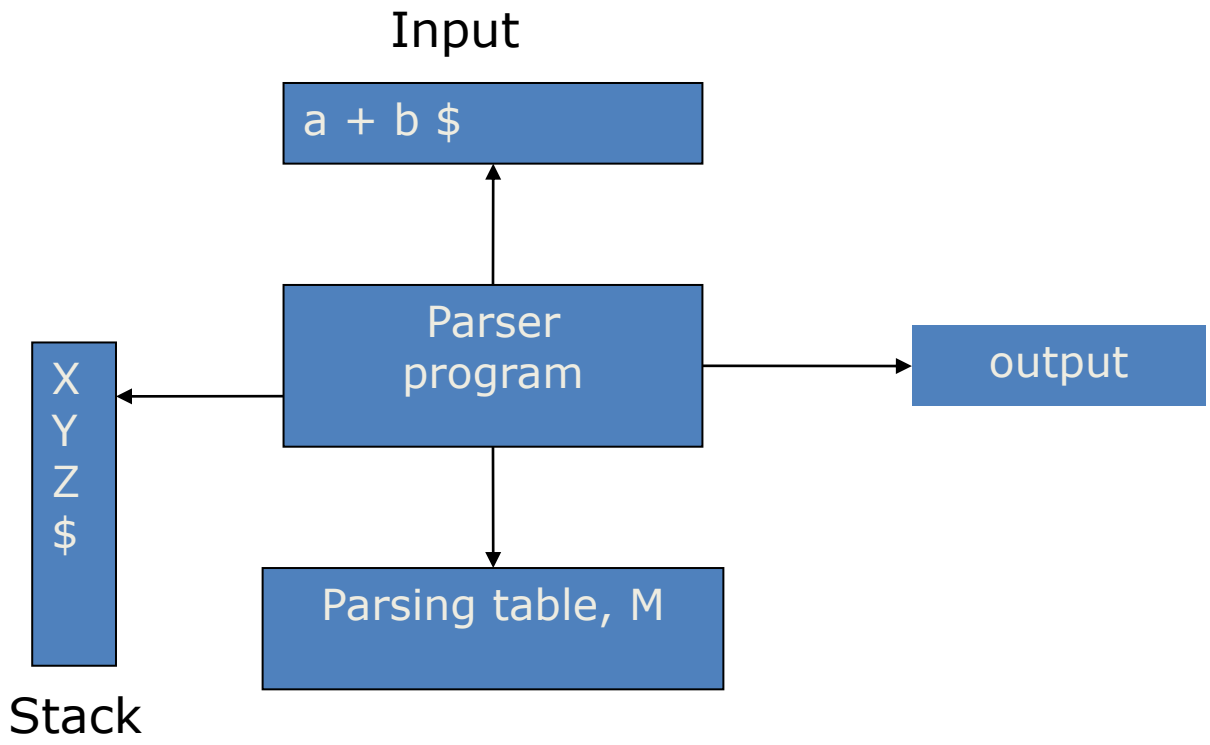
(Hint: follow the previous steps)

Non-recursive Predictive Parsing

- It is possible to build a non-recursive parser by explicitly maintaining a stack.

- This method uses a parsing table that determines the next production to be applied.

Predictive Parsing (cont'd)



Predictive Parsing (cont'd)

- The input buffer contains the string to be parsed followed by $ (the right end marker)
- The stack contains a sequence of grammar symbols with $ at the bottom.
  - Initially, the stack contains the start symbol of the grammar followed by $.
- The parsing table is a two dimensional array M[A, a] where **A** is a non-terminal of the grammar and **a** is a terminal or $

Predictive Parsing (cont'd)

- The parser program behaves as follows:
  - The program always considers x, the symbol on top of the stack and a, the current input symbol.
  - There are three possibilities:
    - x = a = $: the parser halts and announces successful completion of parsing
    - x = a ≠ $: the parser pops x off the stack and advances the input pointer to the next symbol
    - x is a non-terminal : the program consults entry M[x, a] which can be an x-production or an error entry.

Predictive Parsing (cont'd)

- If M[x, a] = {x → uvw}, x on top of the stack will be replaced by uvw (u at the top of the stack). As an output, any code associated with the x-production can be executed.

- If M[x, a] = error, the parser calls the error recovery method.

Predictive Parsing (cont'd)

- Example:

  G:

  E → TR

  R → +TR

  R → -TR

  R → λ

  T → 0|1|…|9

Predictive Parsing (cont'd)

- Parsing table

x | a	0	1	…	9	+	-	$
E	E → TR	E → TR	…	E → TR	Error	Error	Error
R	Error	Error	…	Error	R→+TR	R →-TR	R → λ
T	T → 0	T → 1	…	T → 9	Error	Error	Error

Exercise: Parse **1+2**

Input	Stack
1+2$	E$
1+2$	TR$
1+2$	1R$
+2$	R$
+2$	+TR$
2$	TR$
2$	2R$
$	R$
$	$

Constructing a predictive parsing table

- Makes use of two functions: first and follow

First

- First(α) = set of terminals that begin the strings derived from α.
- If α => λ in zero or more steps, λ is in first(α)

Constructing a predictive parsing table (cont'd)

- First(x), where x is a grammar symbol, can be found using the following rules:
  - If x is a terminal, then first(x) = {x}
  - If x is a non-terminal
    - If x → λ is a production, then add λ to first(x)
    - For each production x → $y_1y_2...y_k$, place a in first(x) if for some i, a ∈ first(y_i) and λ ∈ first(y_j), for 1<j<i

    If λ ∈ first(y_j), for j=1, …,k then λ ∈ first(x)

Constructing a predictive parsing table (cont'd)

- For any string $y = x_1x_2...x_n$
  - Add all non-λ symbols of first(x_1) in first(y)
  - Add all non-λ symbols of first(x_i) for $i \neq 1$ if for all $j<i$, $\lambda \in$ first(x_j)
  - $\lambda \in$ first(y) if $\lambda \in$ first(x_i) for $1<=i<=n$

Example

E→TE'

E'→+TE'|λ

T→FT'

T'→*FT'|λ

F→(E)|id

Then:

First(F)=First(T)=First(E)={(,id}

First(E')={+, λ}

First(T')={ *,λ}

Constructing a predictive parsing table (cont'd)

- Example

G: E → TR

R → +TR

R → -TR

R → λ

T → 0|1|…|9

Find:

First(TR)

First(+TR)

First(-TR)

First(0)

Constructing a predictive parsing table (cont'd)

Follow

- Follow(A) = set of terminals that can appear immediately to the right of A in some sentential form.
- Place $ in follow(A), where A is the start symbol
- If there is a production B→αAβ, then everything in first(β), except λ, should be added to follow(A).
- If there is a production B →αA or B → αAβ and λ ∈ first(β), all elements of follow(B) should be added to follow(A).

Constructing a predictive parsing table (cont'd)

- Example

G: E → TR

R → +TR

R → -TR

R → λ

T → 0|1|…|9

Find:

Follow(E)

Follow(R)

Follow(T)

$S \rightarrow ABCDE$

$A \rightarrow a/\epsilon$

$B \rightarrow b/\epsilon$

$C \rightarrow c$

$D \rightarrow d/\epsilon$

$E \rightarrow e/\epsilon$

$S \rightarrow Bb/Cd$

$B \rightarrow aB/\epsilon$

$C \rightarrow cC/\epsilon$

$E \rightarrow TE'$

$E' \rightarrow +TE'/\epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT'/\epsilon$

$F \rightarrow id/(E)$

$S \rightarrow ACB/CbB/Ba$

$A \rightarrow da/BC$

$B \rightarrow g/\epsilon$

$C \rightarrow h/\epsilon$

Construction of the table

- For each production of the form A → α of the grammar do
  - For each terminal a in first(α), add A → α to M[A, a]
  - If λ ∈ first(α), add A → α to M[A, b] for each b in follow(A)

  If λ ∈ first(α) and $ ∈ follow(A), add A → α to M[A, $]
- Make each undefined entry of M be an error.
- Exercise: construct the parsing table for the grammar in the previous example.

- A grammar for which the parsing table does not have a multiply-defined entries is called an LL(1) grammar.

Determining LL(1) grammars:

- A grammar is LL(1) iff for each pair of A-productions $A \rightarrow \alpha \mid \beta$
  - For no terminal a do α and β can derive strings beginning with a
  - At most one of α and β can derive to an empty string (λ)
  - If $\beta \rightarrow \lambda$ then α does not derive to any string beginning with a terminal in follow(A)

LL(1) Grammars

- The first “L” in LL(1) stands for scanning the input from left to right, the second “L” for producing the leftmost derivation, and “1” for using one input symbol of lookahead at each step to make parsing action decisions

- A grammar G is in LL(1) if and only if whenever A→α | β are two distinct productions of G, the following conditions hold:
  1. For no terminal a do both α and β derive strings beginning with a
  2. At most one of α and β can derive the empty string
  3. If β => λ in zero or more steps, then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if α => λ in zero or more steps, then β does not derive any string beginning with a terminal in FOLLOW(A)

- The first two conditions are equivalent to the statement that FIRST(α) and FIRST(β) are disjoint sets
- The third condition is equivalent to stating that if λ is in FIRST(β), then FIRST(α) and FOLLOW(A) are disjoint sets, and likewise if λ is in FIRST(α)
- Note: LL(1) grammar is a grammar for which the parsing table does not have a multiply-defined entries
- no left-recursive or ambiguous grammar can be LL(1)

LL(1) Grammars (cont'd)

- Example: Let G be given by:

  S → tSS' | a | Ea

  S' → eS | λ

  E → b

  Is G an LL(1) grammar?

Error recovery in predictive parsing

- An error is detected in predictive parsing when
  - the terminal on top of the stack does not match the next input symbol

    or
  - there is a non-terminal A on top of the stack and a is the next input symbol and M[A, a] = error.

Error recovery (cont'd)

- Panic mode error recovery method
  - Synchronization tokens

1. If A is the non-terminal on top of the stack, all symbols of follow(A) can be in the set of synchronization symbols for that non-terminal.

   Drop tokens until an element in follow(A) is seen and pop A from the stack, most likely parsing can continue.

2. If the symbol in follow(A) is missing, then the next constructs will also be skipped until the synchronization token of a similar construct is obtained.

Read more

Bottom-up parsing (LR(k) parsing)

- a parsing method that consists of producing the parse tree from the leaves to the root.
- We will see a general style of bottom-up parsing called **shift-reduce parsing**.

• Reductions

- We can think of shift-reduce parsing process as one of "reducing" a string w to the start symbol of the grammar
- At each reduction step a particular substring matching the right side of a production (body) is replaced by the symbol of the left of that production (head), and if the substring is chosen correctly at each step, a rightmost derivation is traced out in reverse
- The key decisions in bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds

Example:

S → aABe

A → Abc | b

B → d string: abbcde

Handle Pruning

- Bottom-up parsing during a left-to-right scan of the input constructs a rightmost derivation in reverse
- Informally, a “handle” is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation

Example: Consider the following grammar

E→E+T | T

T→T*F | F

F→(E) | **id**

- The following steps show the reductions of **id*id** to E
- **id*id**, F***id**, T***id**, T*F, T, E

LR(k) parsing

S → aABe

A → Abc | b

B → d string: abbcde

A bottom-up derivation of abbcde:

abbcde ← aAbcde ← aAde ← aABe ← S

- At each step, we have to find α such that α is a substring of the sentential form and replace α by A, where A → α

Handle (cont'd)

- Definition:
  A handle of a right sentential form γ is a production A $\rightarrow$ β and a position of γ where the string β may be found and replaced by A to produce the previous sentential form in the rightmost derivation of γ.

- That is, if S $\Rightarrow$ αAw $\Rightarrow$ $\alpha\beta$w, then A $\rightarrow$ β in the position following α is a handle of $\alpha\beta$w.

Handle pruning

- Replacing β by A in the string αβw using the production A → β is called **handle pruning**.
- Therefore, a rightmost derivation can be obtained by handle pruning.
- Example: Let G be given by:

  E → E + E | E*E | (E) | id

  Find a rm derivation for: id + id * id

Handle pruning (cont'd)

Right sentential form	**Reducing production**
<u>id</u> + id * id	E → id
E + <u>id</u> * id	E → id
E + E * <u>id</u>	E → id
E + <u>E * E</u>	E → E * E
<u>E + E</u>	E → E + E
E	

Shift-Reduce Parsing

- In LR parsing the two major problems are:
  - locate the substring that is to be reduced
  - locate the production to use for the reduction
- Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed

- As we shall see, the handle always appears at the top of the stack just before it is identified as the handle
- $ is used to mark the bottom of the stack and also the right end of the input
- Initially, the stack is empty, and the string w is on the input as follows:
- Stack Input
- $ w$

Stack implementation of shift/reduce parsing (cont'd)

- A shift/reduce parser operates by shifting zero or more input into the stack until the right side of the handle (β) is on top of the stack.
- The parser then replaces β by the non-terminal of the production.
- This is repeated until the start symbol is in the stack and the input is empty, or until error is detected.

Stack	Input
E$	$

- Upon entering this configuration, the parser halts and announces successful completion of parsing

Stack implementation of shift/reduce parsing (cont'd)

- Four actions are possible:
  - shift: the next input is shifted on to the top of the stack
  - reduce: the parser knows the right end of the handle is at the top of the stack. It should then decide what non-terminal should replace that substring
  - accept: the parser announces successful completion of parsing
  - error: the parser discovers a syntax error

- Example: Consider the following grammar

E → E + E | E*E | (E) | id

Stack implementation of shift/reduce parsing (cont'd)

- Example: operations of a shift/reduce parser

stack	input	action
$	id + id * id $	shift
id $	+ id * id $	reduce by E→ id
E $	+ id * id $	shift
+ E $	id * id $	shift
id + E $	* id $	reduce by E → id
E + E $	* id $	shift
* E + E $	id $	shift
id * E + E $	$	reduce by E → id
E * E + E $	$	reduce by E → E * E
E + E $	$	reduce by E → E + E
E $	$	accept

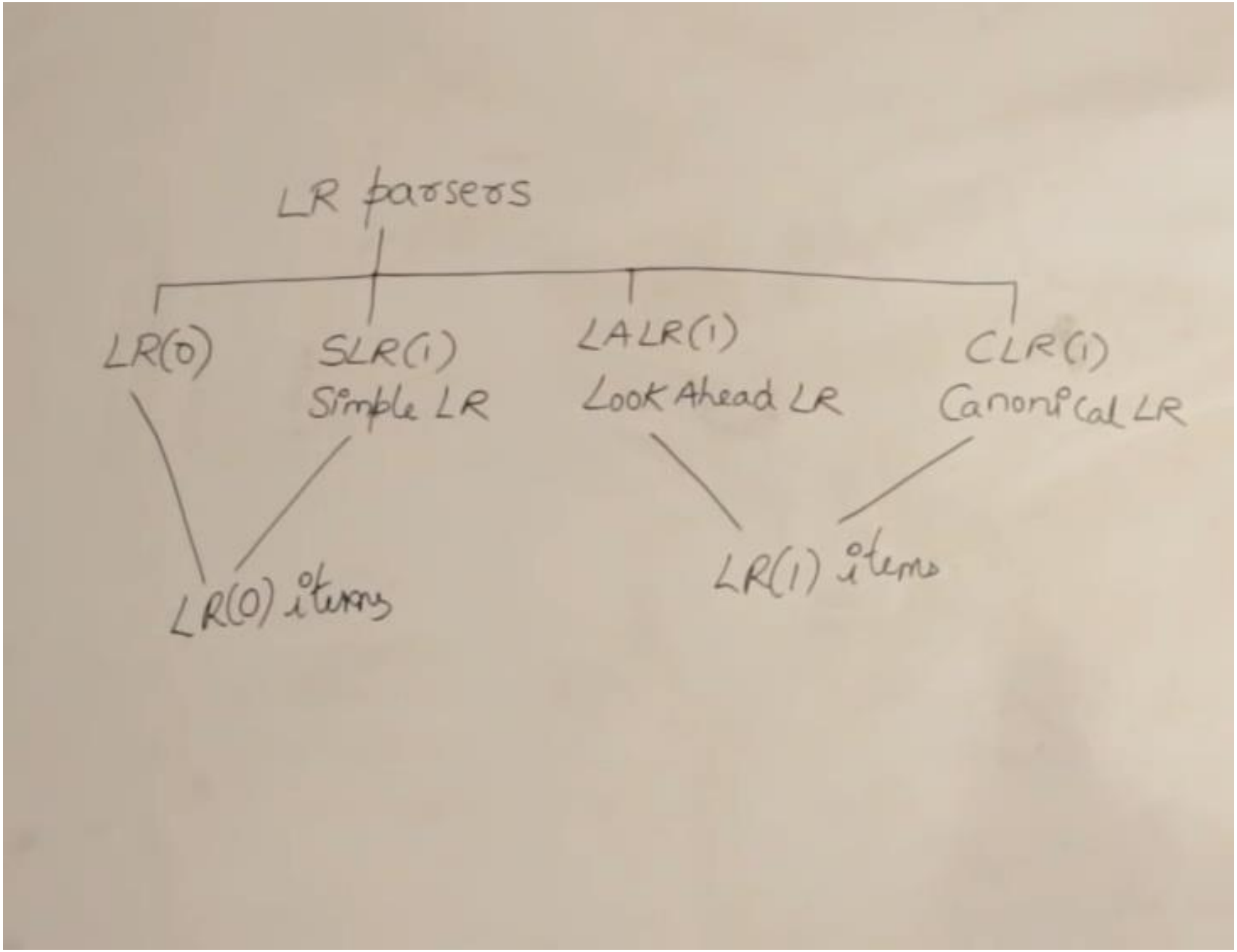
Conflict during shift/reduce parsing

- There are two types of conflicts in shift/reduce parsing:
  - **shift/reduce conflict**: the parser knows the entire stack content and the next k symbols but cannot decide whether it should shift or reduce.
  - **reduce/reduce conflict**: the parser cannot decide which of the several productions it should use for a reduction.

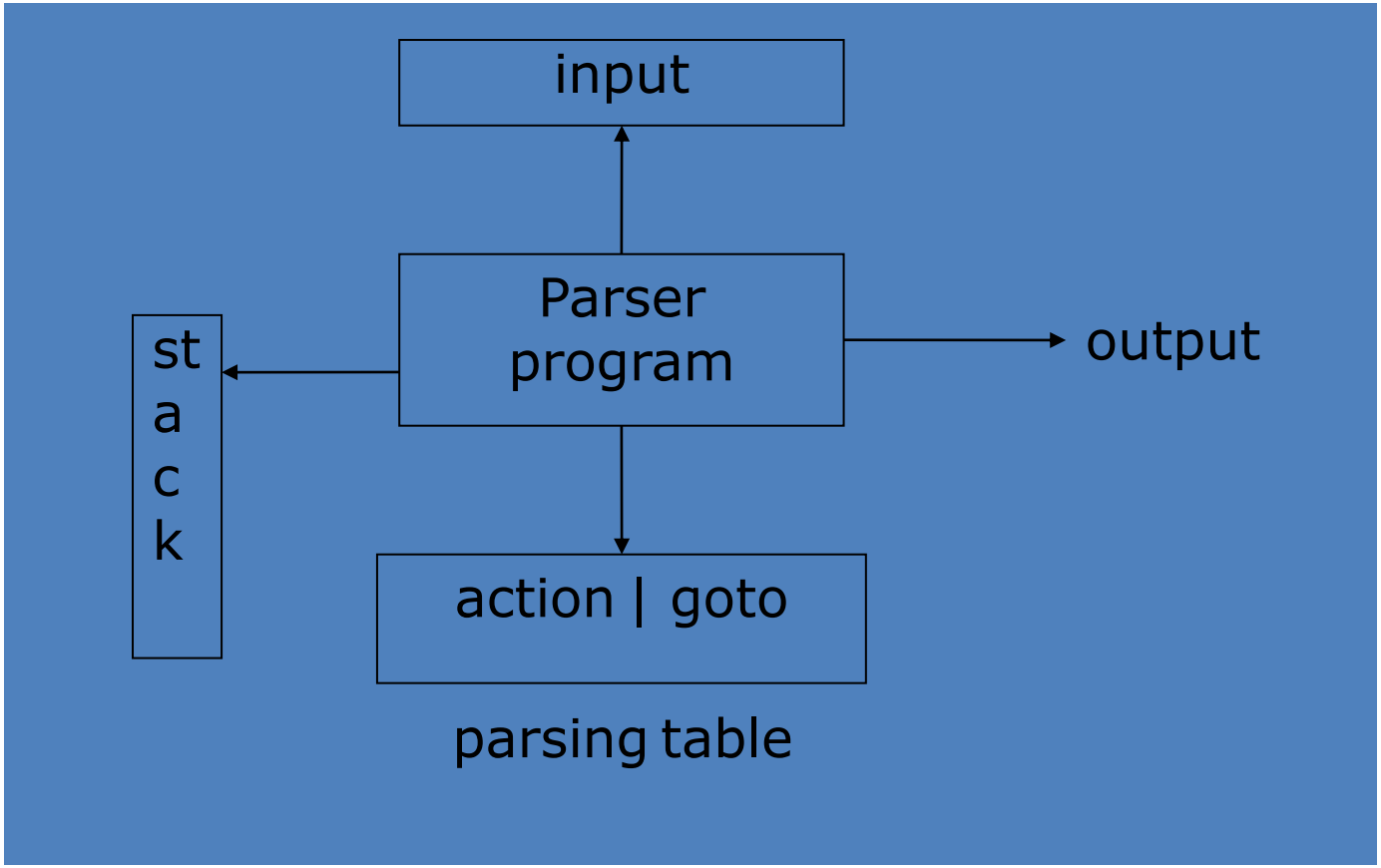
LR(k) Grammars

- Grammars for which we can construct an LR(k) parsing table are called LR(k) grammars.
- Most of the grammars that are used in practice are LR(1).
- k stands for the number of look ahead symbols used by the parser.

- The L” is for left-to-right scanning of the input, the “R” for constructing a rightmost derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decisions


LR parsers
LR(0)
SLR(1)
Simple LR
LALR(1)
Look Ahead LR
CLR(1)
Canonical LR
LR(0) items
LR(1) items

LR parsers


input
Parser program
output
st a c k
action | goto
parsing table

LR parsers (cont'd)

- The LR(k) stack stores strings of the form: $S_0X_0S_1X_1…X_mS_m$ where
  - S_i is a new symbol called state that summarizes the information contained in the stack
  - S_m is the state on top of the stack
  - X_i is a grammar symbol

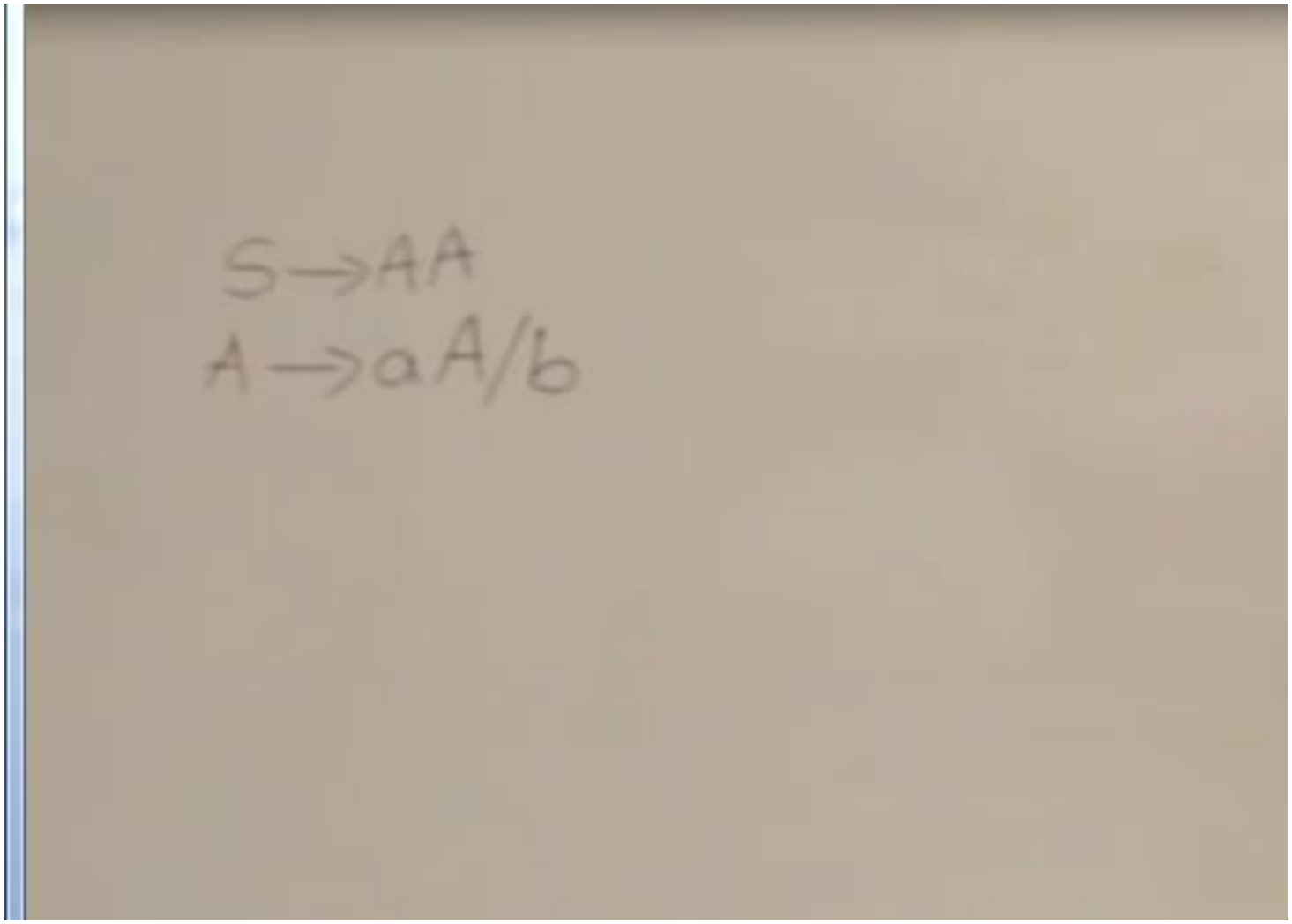
LR parsers (cont'd)

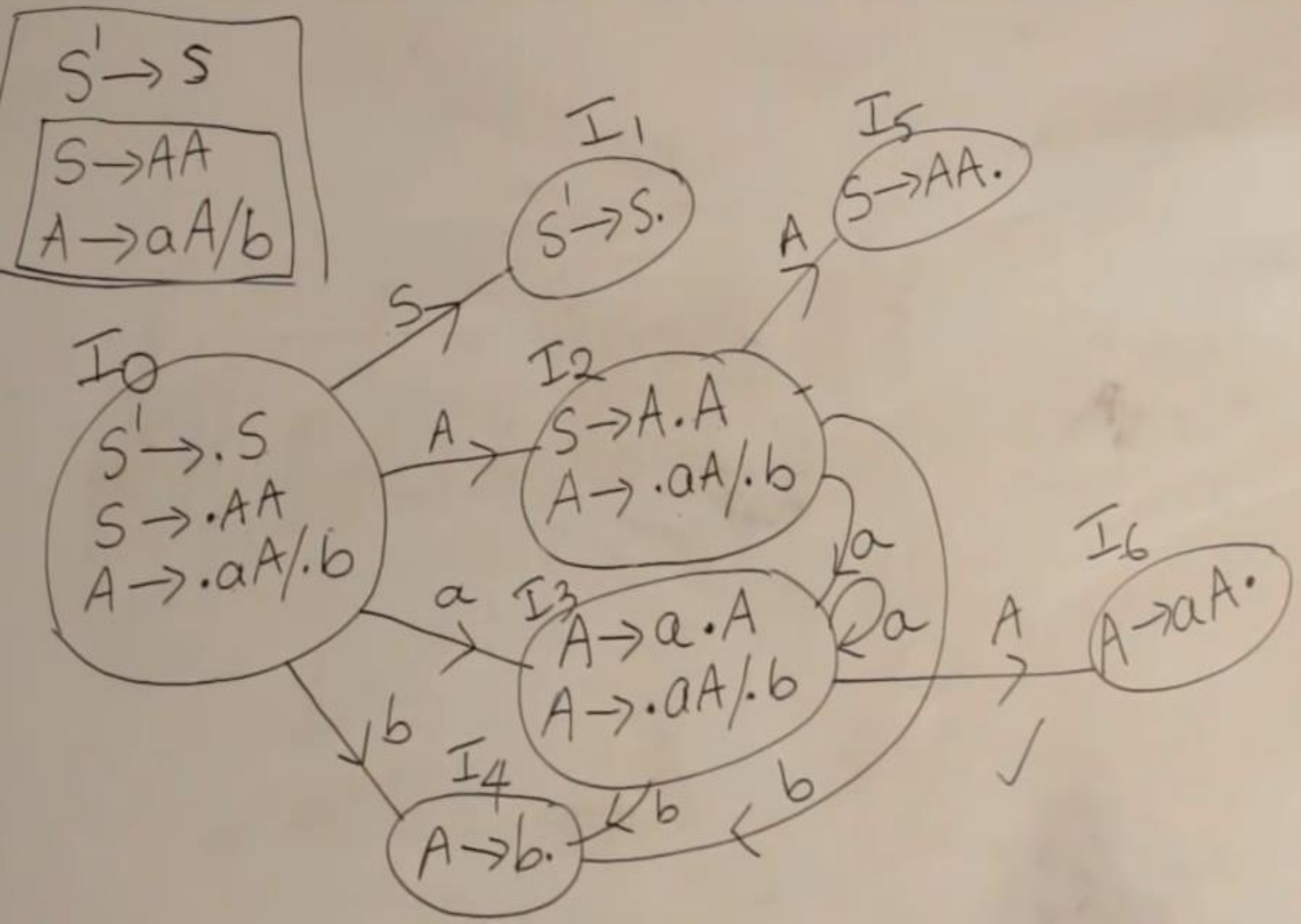
- The parser program decides the next step by using
  - the top of the stack (S_m),
  - the input symbol (a_i),
  - the parsing table (which has two sections : action section and goto section)

LR parsers (cont'd)

- Action[S_m, a_i] can be:
  - Action[S_m, a_i] = shift s:
    - moves the input cursor past a_i
    - puts a_is in the stack
  - Action[S_m, a_i] = reduce A → β:
    - pops the first 2r symbols off the stack, where r = /β/ (at this point, S_{m-r} will be the state on top of the stack).
    - pushes A and s on top of the stack where s = goto[S_{m-r}, A].

    The input buffer is not modified.
- Note: s_o is on top of the stack at the beginning of the parsing.


S→AA
A→aA/b


S'→S
S→AA
A→aA/b
I0
S'→.S
S→.AA
A→.aA/.b
I1
S'→S.
I2
S→A.A
A→.aA/.b
I3
A→a.A
A→.aA/.b
I4
A→b.
I5
S→AA.
I6
A→aA.
S
A
a
b
a
b
a
A
b

LR(0)

	action			Goto	
	a	b	$	A	S
0	S_3	S_4		2	1
1			accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	γ_3	γ_3	γ_3		
5	γ_1	γ_1	γ_1		
6	γ_2	γ_2	γ_2		

LR parsers (cont'd)

- Example:

  Let G1 be:

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

state	action						goto		
	id	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
0	s5			s4			1	2	3
1		s6				accept			
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			

Legend: si = shift to state i Rj = reduce by production j

LR parsers (cont'd)

- Apply shift/reduce parsing to parse the input string w = id * id + id using the given parsing table

How a shift/reduce parser parses the input **id*id+id**

Stack	**Input**	**Action**
0	id * id + id $	shift 5
0 id 5	* id + id $	reduce 6 (F→id)
0 F 3	* id + id $	reduce 4 (T → F)
0 T 2	* id + id $	shift 7
0 T 2 * 7	id + id $	shift 5
0 T 2 * 7 id 5	+ id $	reduce 6 (F → id)
0 T 2 * 7 F 10	+ id $	reduce 3 (T → T * F)
0 T 2	+ id $	reduce 2 (E → T)
0 E 1	+ id $	shift 6
0 E 1 + 6	id $	shift 5
0 E 1 + 6 id 5	$	reduce 6 (F → id)
0 E 1 + 6 F 3	$	reduce 4 (T → F)
0 E 1 + 6 T 9	$	reduce 1 (E → E + T)
0 E 1	$	accept

Constructing LR parsing tables

- There are three methods for constructing LR parsing tables:
  - SLR (Simple LR) method
  - LALR method
  - Canonical LR method

Constructing SLR parsing tables

- This method is the simplest of the three methods used to construct an LR parsing tables.
- It is called SLR (simple LR) because it is the easiest to implement. However, it is also the weakest in terms of the number of grammars for which it succeeds.
- A parsing table constructed by this method is called SLR table.
- A grammar for which an SLR table can be constructed is said to be an SLR grammar.

SLR tables (cont'd)

- **LR (0) item**
  - An LR (0) item (item for short) is a production of a grammar G with a dot at some position of the right side.
  - Ex. A → XYZ has the ff items:

    A → . X Y Z , A → X . Y Z

    A → X Y . Z , A → X Y Z.
  - For the production A → λ we only have one item: A →.
  - An item indicates what part of a production we have seen and what we expect to see.

SLR tables (cont'd)

- The central idea in SLR method is to construct, from the grammar, a DFSA to recognize viable prefixes.
- A viable prefix is a prefix of a right sentential form that can appear on the stack of a shift/reduce parser.
- If we have a viable prefix in the stack it is possible to have inputs that will reduce to the start symbol.
- Otherwise, we can never reach the start symbol; therefore we have to call the error recovery procedure.

SLR tables (cont'd)

- The closure operation
  - If I is a set of items of G, then Closure (I) is the set of items constructed by two rules:
    - Initially, every item in I is added to Closure (I)
    - If A → α.Bβ is in Closure of (I) and B→ γ is a production, then add B → .γ to I. This rule is applied until no more new item can be added to Closure (I)

SLR tables (cont'd)

- Example G1':

E' → E

E → E + T

E → T

T → T * F

T → F

F → (E)

F → id

I = {[E' → .E]}

Find Closure (I)

Closure (I) = {
[E' → .E], [E → .E + T],
[E → .T], [T → .T * F],
[T → .F], [F → .(E)],
[F → .id] }

SLR tables (cont'd)

- The Goto operation
  - Where I is a set of items and X is a grammar symbol, Goto (I, X) is defined as the closure of all items [A → αX.β] such that [A → α.Xβ] is in I.
  - Example: I = {[E' → E.], [E → E . + T]}

  Then Goto (I, +) = ?

  { [E → E + . T], [T → .T * F], [T → .F],
  [F → .(E)] [F → .id] }

SLR tables (cont'd)

- The set of Items construction
  - the canonical collection of sets of LR (0) items for augmented grammar G' can be constructed as follows:

```
Procedure Items (G');
Begin
    C := {Closure ({[S' → . S]})}
    Repeat
      For Each item of I in C and each grammar symbol X  such that Goto
    (I, X) is not empty and not in C do
                    Add Goto (I, X) to C;
      Until no more sets of items can be added to C
End
```

SLR tables (cont'd)

- Ex: Construct the set of Items for the following augmented grammar

E' → E

E → E + T

E → T

T → T * F

T → F

F → (E)

F → id

SLR tables (cont'd)

- Valid items
  - We say an item A → β_1. β_2 is valid viable prefix of $\alpha\beta_1$ if there is derivation S' drives in zero or more steps to $\alpha\beta w$ and $\alpha\beta w$ drives in zero or more steps to $\alpha\beta_1\beta_2 w$.
  - If A → $\beta_1.\beta_2$ is valid, it tells us to shift or reduce when $\alpha\beta_1$ is on top of the stack.

- Theorem
  - The set of valid items embodies all the useful information that can be gleamed from the stack.

SLR table construction method

1. Construct C = {I_0, I_1, ..., I_N} the collection of the set of LR (0) items for G'.
2. State i is constructed from I_i and
   1. If [A → α.aβ] is in I_i and Goto (I_i, a) = I_j (a is a terminal) then action [i, a]=shift j
   2. If [A →α.] is in I_i then action [i, a] = reduce A→α for a in Follow (A) for A ≠ S'
   3. If [S' → S.] is in I_i then action [i, $] = accept.

   If no conflicting action is created by 1 and 2, the grammar is SLR (1); otherwise it is not.
3. For all non-terminals A, if Goto (I_i, A) = I_j then Goto [i, A] = j
4. All entries of the parsing table not defined by 2 and 3 are made error
5. The initial state is the one constructed from the set of items containing [S' → .S]

SLR tables (cont'd)

- Example1: Construct the SLR parsing table for G1'

G1':

E' → E

E → E + T

E → T

T → T * F

T → F

F → (E)

F → id

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

SLR tables (cont'd)

- Example2: Construct the SLR parsing table for G2'

G2':

S' → S

S → L = R

S → R

L → *R

L → id

R → L

Follow (S) = {$}
Follow (R) = {$, =}
Follow (L) = {$, =}

- Given G:

  A → B ; A

  A → B

  B → C D

  C → array E of C

  C → int

  D → id , D

  D → id

  E → num

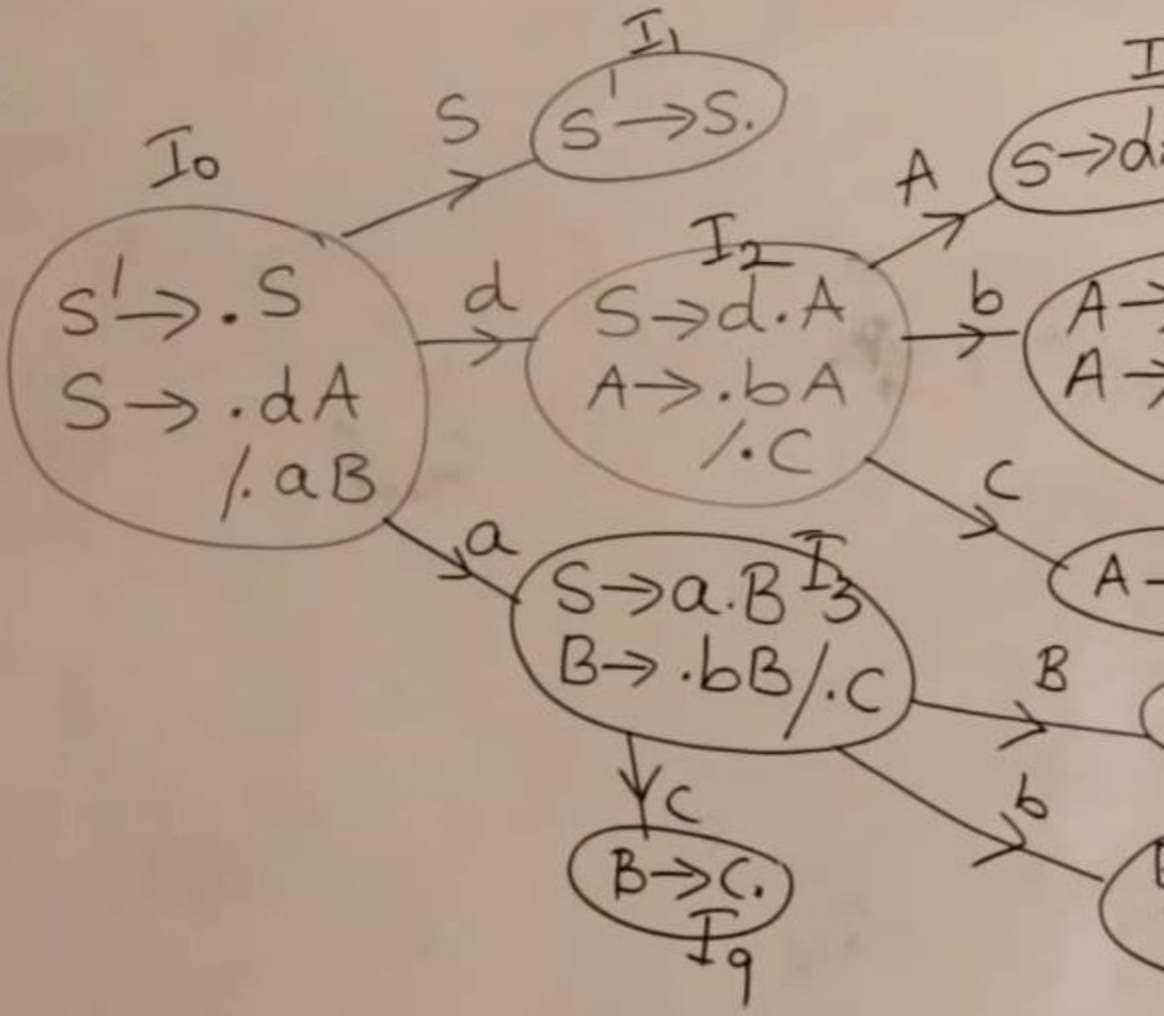
-Construct SLR(1) table for the given grammar and parse the string **array 8 of int x1, x2; array 7 of 9 x3**

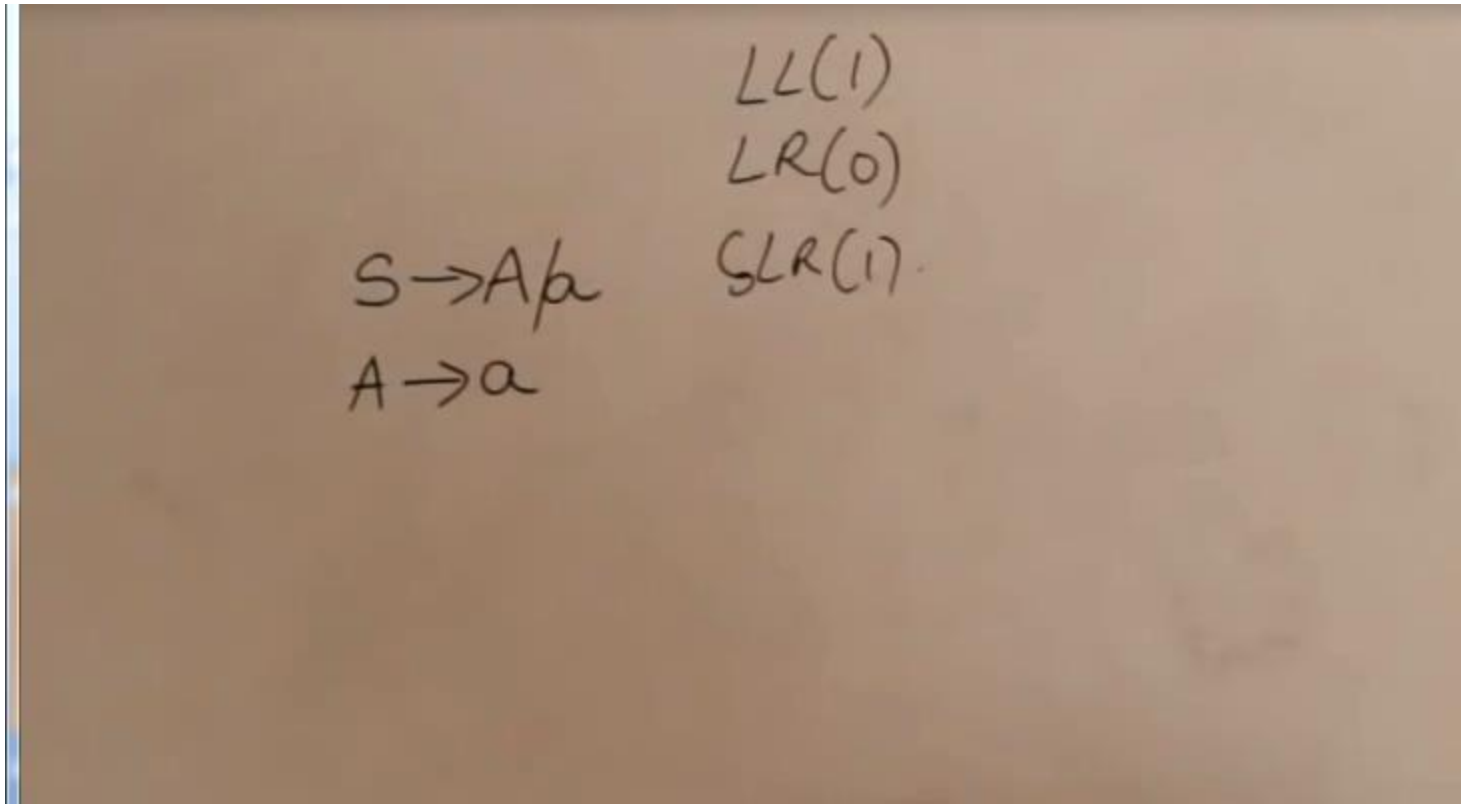
(i) LL(1)
(ii) LR(0)
(iii) SLR(1)

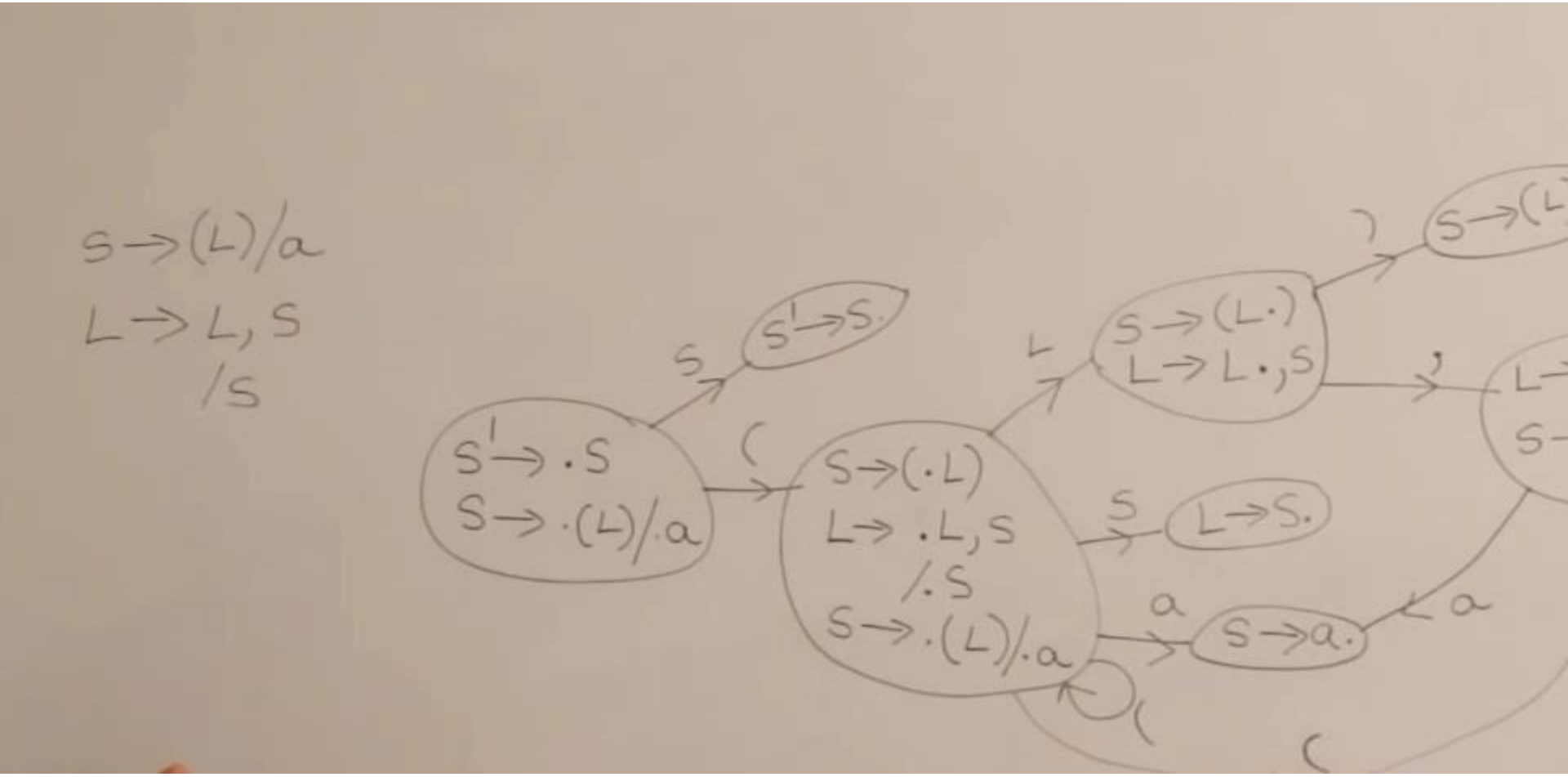
$S \to dA \mid aB$

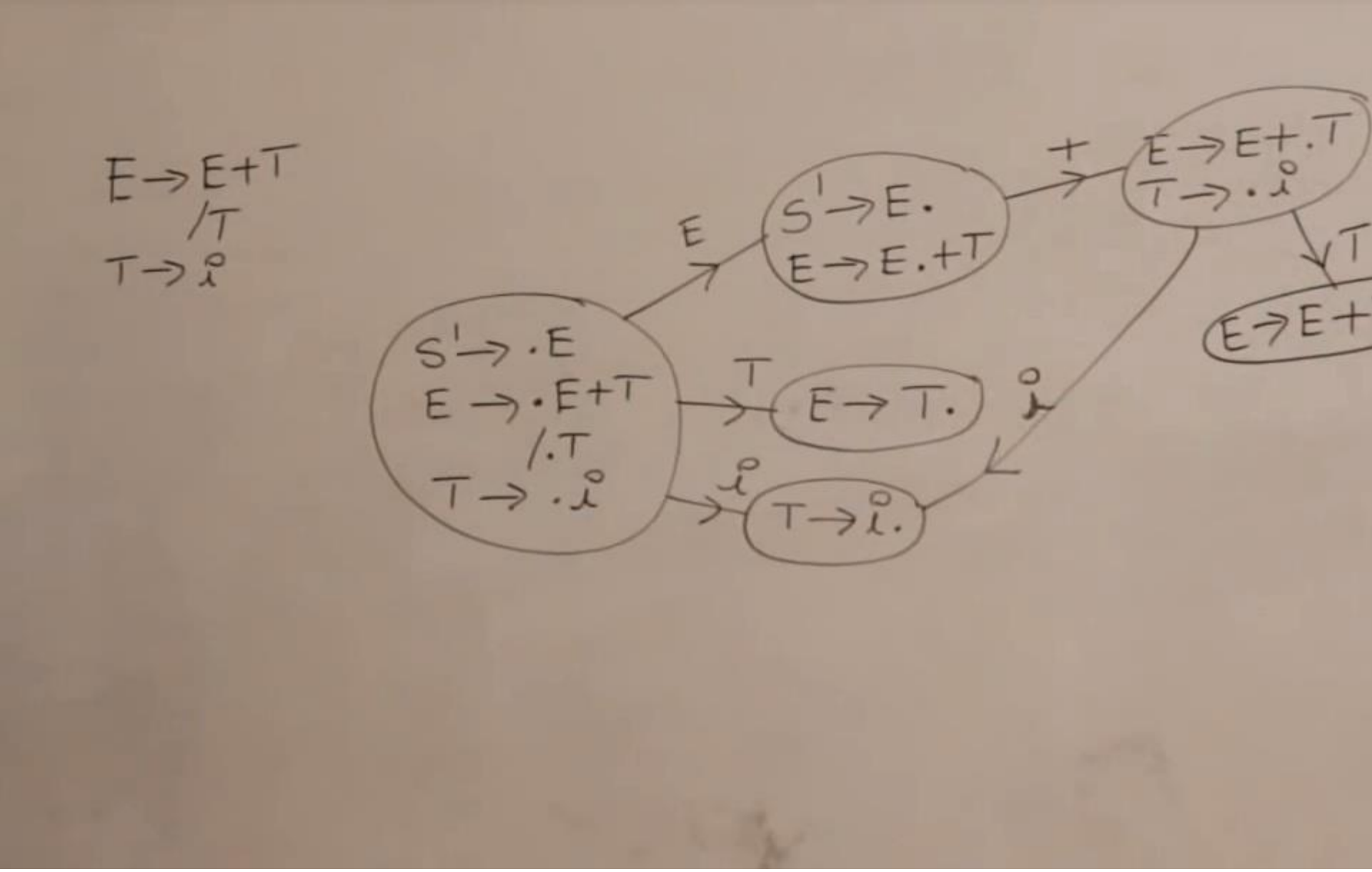
$A \to bA \mid c$

$B \to bB \mid c$




LL(1)
LR(0)
SLR(1)
S→Aa
A→a


S→(L)/a
L→L,S
/S
S'→.S
S→.(L)/.a
S
S'→S.
(
S→(.L)
L→.L,S
/.S
S→.(L)/.a
L
S→(L.)
L→L.,S
S
L→S.
a
S→a.
)
S→(L)
,
a
(


E→E+T
/T
T→i
S'→.E
E→.E+T
/.T
T→.i
E
T
i
S'→E.
E→E.+T
E→T.
T→i.
+
E→E+.T
T→.i
T
E→E+

Canonical LR parsing

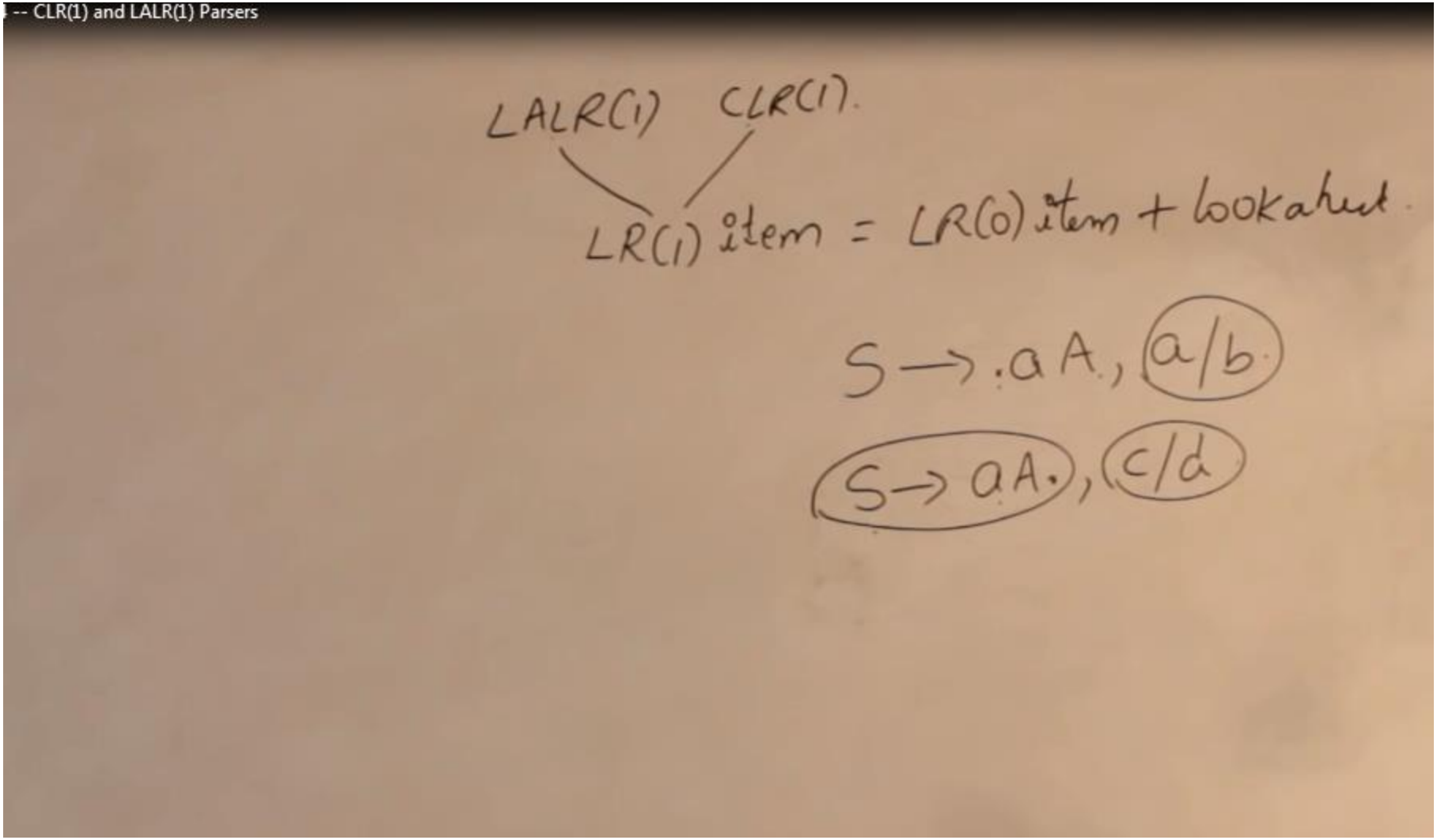
- The states produced using LR (0) items do not hold enough inf^n.
- It is possible to hold more inf^n in the state to rule out some invalid reductions.
- By splitting states when necessary, we indicate which symbol can exactly follow a handle.
- Consider the previous example (G2').

Follow(S)={$};Follow(R)={$,=};Follow(L)={$,=}

We have shift/reduce conflict since

= is in Follow (R) and R → L. is in I2 and Goto (I2, =) = I6

action[2, =] = shift 6 / reduce R → L


LALR(1) CLR(1).
LR(1) item = LR(0) item + lookahead.
S → .aA., a/b
S → aA., c/d

Canonical LR parsing (cont'd)

- LR(1) item
  - An LR (1) item has the form of
    [A → α.β, a] where a is a terminal or $.

- The functions closure (I), goto(I, X) and Items (G') will be slightly different from those used for the production of an SLR parsing table.

Canonical LR parsing (cont'd)

- The closure operation

  I is a set of LR (1) items, Closure (I) is found using the following algorithm:

  Closure [I] := I

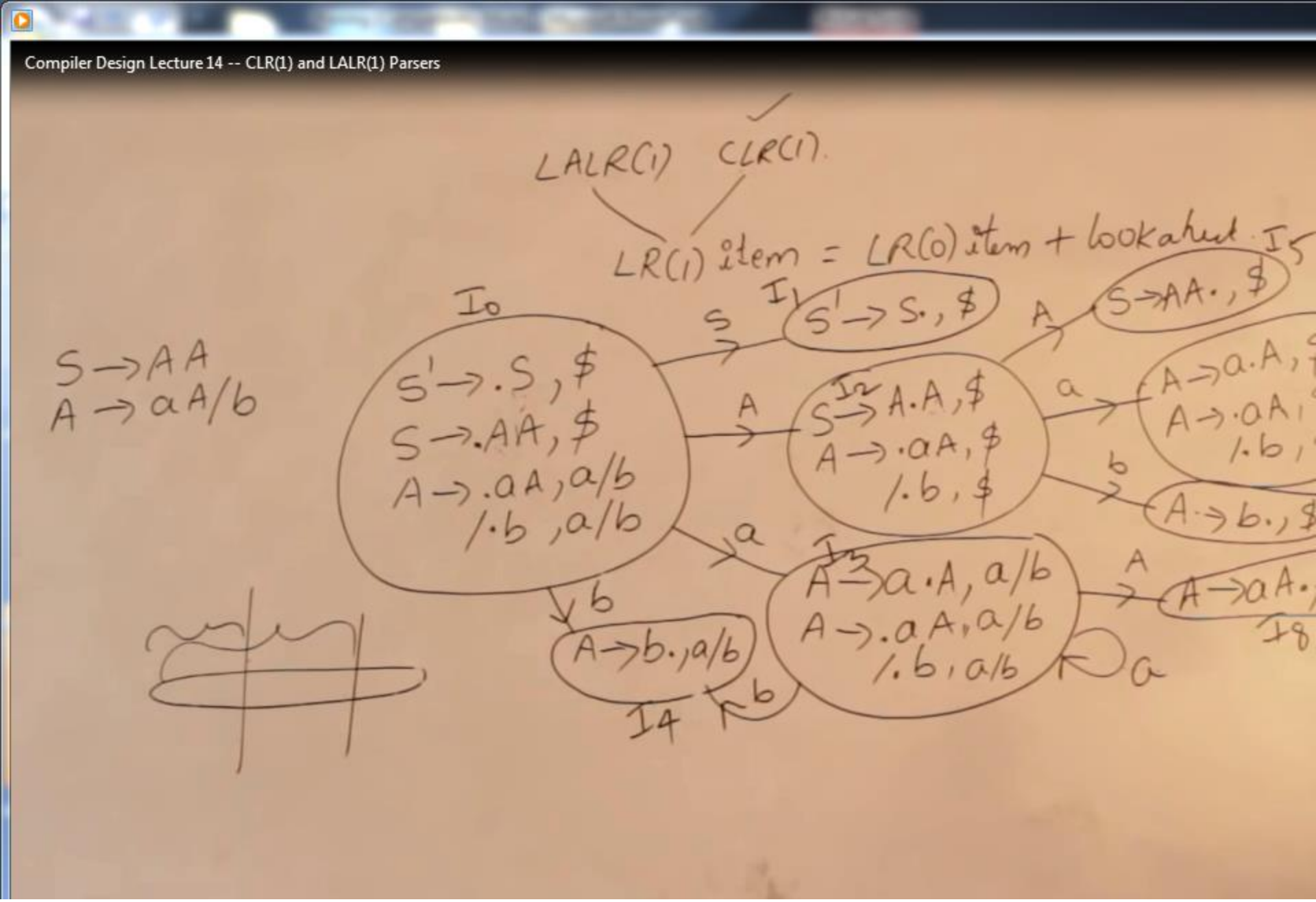
  Repeat

  For each item [A → α. Bβ, a] in closure(I), each production B → γ in G' and each terminal b in First(βa) such that [B→.γ, b] is not in closure (I) do

  add [B → .γ, b] to closure (I)

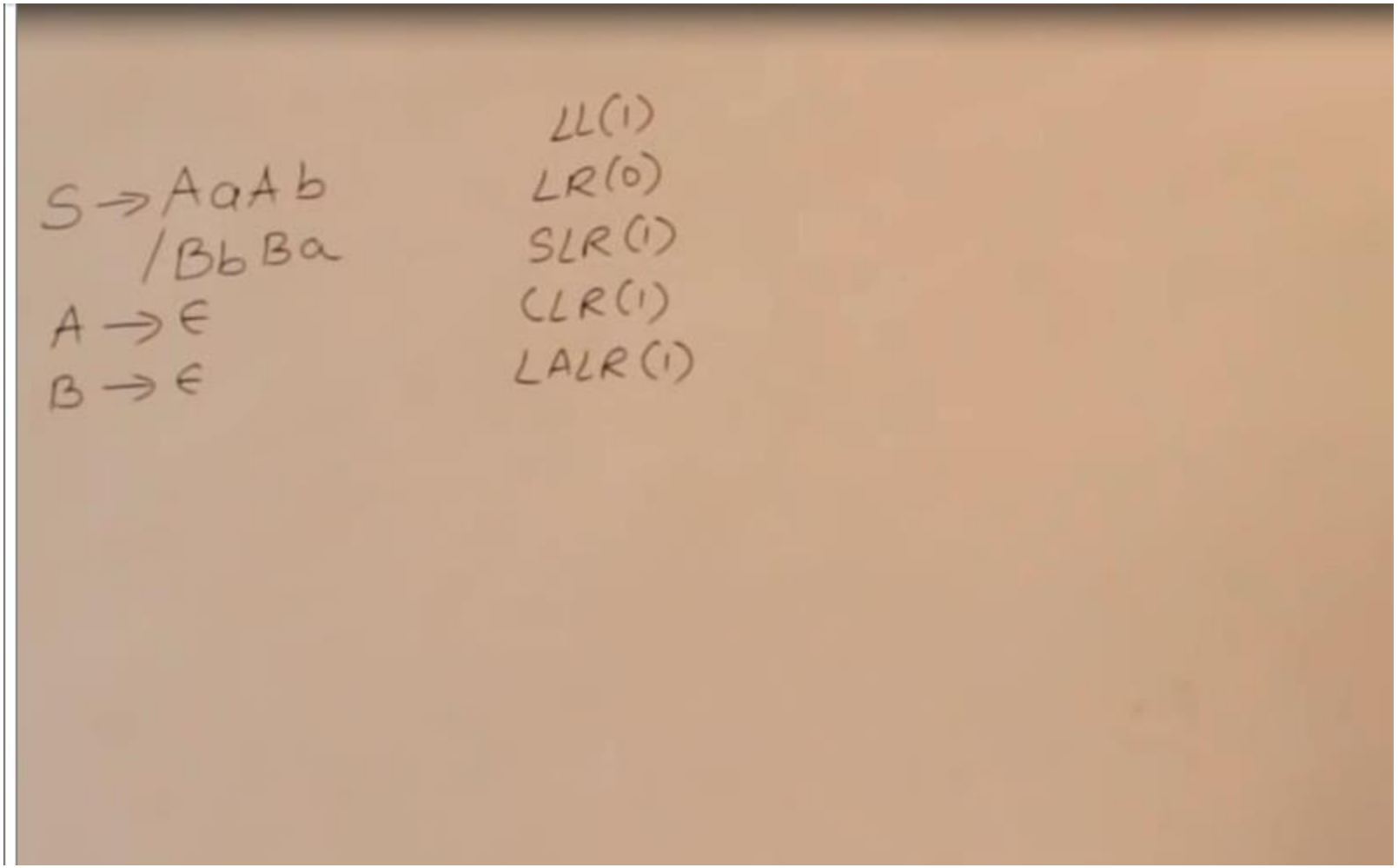
  until no more items can be added to closure (I)

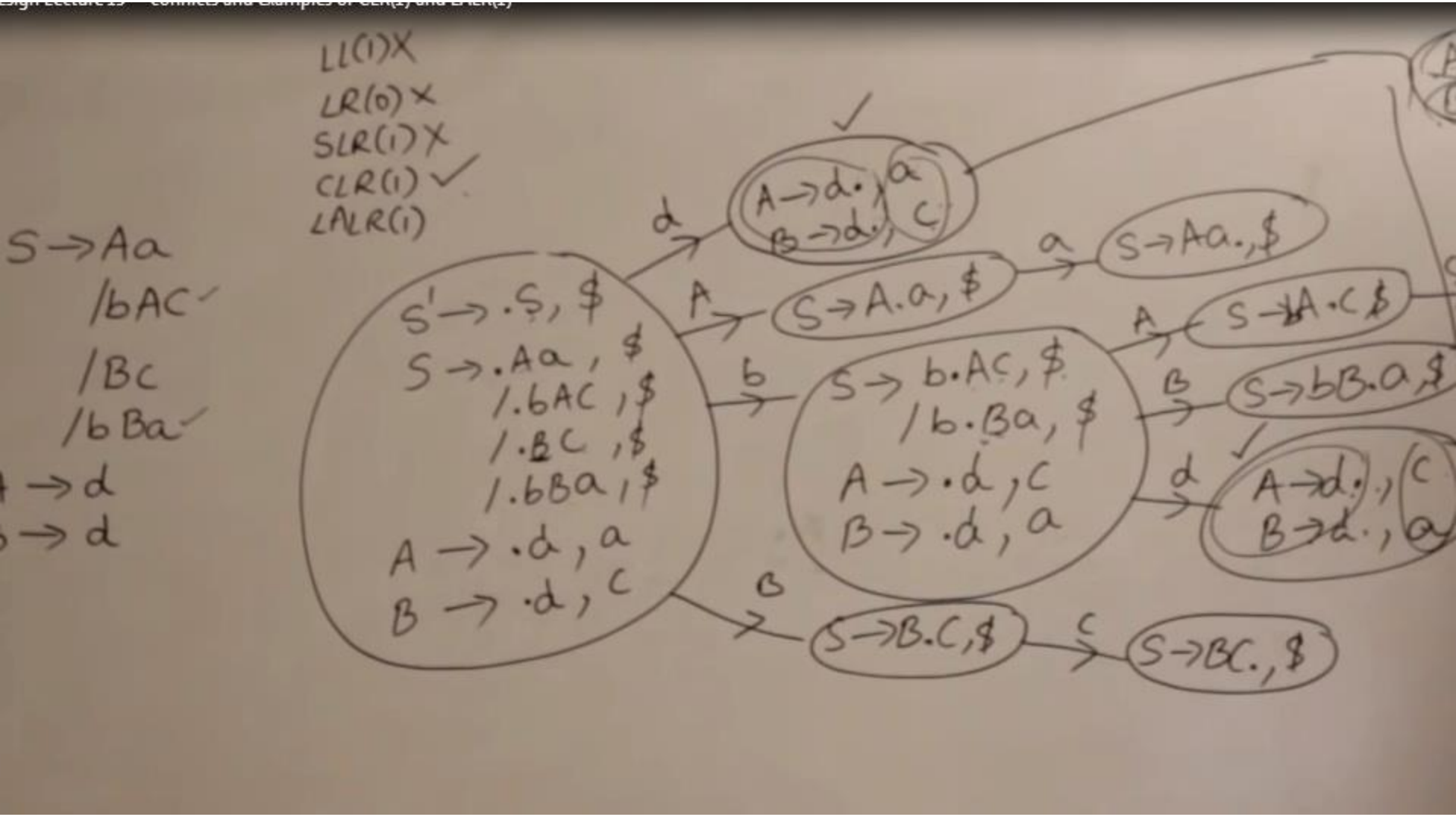
Compiler Design Lecture 14 -- CLR(1) and LALR(1) Parsers



CLR(1) parsing table

	a	b	$	S	A
0	S_3	S_4			2
1					
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		


S → AaAb
/BbBa
A → ∈
B → ∈
LL(1)
LR(0)
SLR(1)
CLR(1)
LALR(1)


LL(1)X
LR(0)X
SLR(1)X
CLR(1)✓
LALR(1)
S→Aa
/bAC
/Bc
/bBa
→d
→d
S'→.S, $
S→.Aa, $
/.bAC, $
/.Bc, $
/.bBa, $
A→.d, a
B→.d, c
d
A→d., a
B→d., c
A
S→A.a, $
a
S→Aa., $
b
S→b.Ac, $
/b.Ba, $
A→.d, c
B→.d, a
A
S→bA.c $
B
S→bB.a $
d
A→d., c
B→d., a
B
S→B.C, $
c
S→Bc., $

Canonical LR parsing (cont'd)

- Example
  - Consider the grammar G2'

    Find Closure {[S' → .S, $]}

    First ($) = {$}

    First (= R $) = {=}

    First () = {=}

  Closure {[S' → .S, $]} =

  {[S' →.S, $], [S →.L = R, $], [S →.R, $],
  [L →.*R, =], [L →.id, =], [R →.L, $],
  [L →.*R, $], [L →.id, $]}

Canonical LR parsing (cont'd)

- The Goto operation
  - Goto (I., X) is defined as the closure of all items [A → αX.β, a] such that [A → α .Xβ, a] is in I.

  Example:

  I_0 = {[S' →.S, $], [S →.L = R, $], [S →.R, $], [L →.*R, =], [L →.id, =], [R →.L, $], [L →.*R, $], [L →.id, $]}

  Find Goto(I_0, S) = {[S' → S., $]}

  Goto(I_0, S) = {[S' → S., $]}

Canonical LR parsing (cont'd)

- The set of Items construction
  - construct C, the canonical collection of sets of LR (1) items

Procedure Items (G');
Begin
  C := {Closure ({[S' → . S, $]})}
  Repeat
    For Each item of I in C and each grammar symbol X such that Goto (I, X) is not empty and not in C do
      Add Goto (I, X) to C;
  Until no more sets of items can be added to C
End

Canonical LR parsing (cont'd)

- Example:
  - Construct C for the grammar G2'

G2':

S' → S

S → L = R

S → R

L → *R

L → id

R → L

Canonical LR parsing (cont'd)

C = {I0, I1, I2, I3, I4, I5, I6, I7, I8, I9}
I0 = {[S' → .S, $], [S → .L = R, $], [S → .R, $], [L → .*R, =|$], [L → .id, =|$], [R → .L, $]}
I1 = goto (I0, S) = {[S' →S., $]}
I2 = goto (I0, L) = {[S →L . = R, $], [R →L., $]}
I3 = goto (I0, R) = {[S →R., $]}
I4 = goto (I0, *) = {[L →* . R, =|$], [L →.*R, =|$], [L →.id, =|$], [R →.L, =|$]}
I5 = goto (I0, id) ={[L →id ., =|$]}
I6 = goto (I2, =) = {[S →L = . R, $], [R →. L, $], [L →.*R, $], [L →.id,$]}
I7 = goto (I4, R) ={[L →* R ., =|$]}
I8 = goto (I4, L) ={[R →L ., =|$]}
goto (I4, *) = I4
goto (I4, id) = I5
I9 = goto (I6, R) ={[S →L = R ., $]}
I10 = goto (I6, L) ={[R →L ., $]}
I11 = goto (I6, *) = {[L →* . R, $], [L →.*R, $], [L →.id, $], [R →.L, $]}
I12 = goto (I6, id) ={[L →id ., $]}
goto (I11, *) = I11
goto (I11, id) = I12
goto (11, L) = I10
I13 = goto (I11, R) = {[L→*R., $]}

Canonical LR parsing (cont'd)

- Construction of LR parsing table

1. Construct C = {I_0, I_1, ... I_n} the collection of LR (1) items for G'
2. State i of the parser is constructed from state I_i. The parsing actions for state i are determined as follows:
   a. If [A → α.aβ, b] is in I_i and Goto (I_i, a) = I_j (a is a terminal) then action [i, a]=shift j
   b. If [A →α., a] is in I_i and A≠ S' then action [i, a] = reduce A→α
   c. If [S' → S., $] is in I_i then action [i, $] = accept.
3. If there is a conflict, the grammar is not LR (1).
4. If Goto (I_i, A) = I_j, then Goto [i, A] = j
5. All entries not defined by (2) and (3) are made error
6. The initial state is the set constructed from the item [S'→.S, $]

Canonical LR parsing (cont'd)

- The table constructed using this method is called the canonical LR(1) parsing table.
- Exercise:
  - Construct the canonical LR(1) parsing table for the grammar G2'

LALR parsing

- The most widely used method since the table obtained using this method is much smaller than that obtained by canonical LR method.
- For example, for a language such as Pascal,
  LALR table will have hundreds of states
  canonical LR table will have thousands of states
- Much more grammars are LALR than SLR, but there are only few grammars that are CLR but not LALR, and these grammars can be easily avoided.

LALR parsing (cont'd)

- Definition
  We call the core of a set of LR (1) items, the set of the first components of the set of LR (1) items.
- Example:

  In the canonical set of LR (1) items of the previous example, both I4 and I11 have the following identical core:

{[L → * . R], [L → .*R], [L → .id], [R → .L]}

LALR parsing (cont'd)

- Method

  Refer to Algorithm 4.11 on Page 238 of the text book.

- Exercise: Construct the LALR parsing table for the grammar G2'.