MODULE 3

Bottom-Up Parsing: Shift Reduce parsing – Operator precedence parsing (Concepts only)

LR parsing – Constructing SLR parsing tables, Constructing, Canonical LR parsing tables and Constructing LALR parsing tables.

3.1 BOTTOM UP PARSING

A bottom-up parse starts with the string of terminals itself and builds from the leaves upward, working backwards to the start symbol by applying the productions in reverse.

Along the way, a bottom-up parser searches for substrings of the working string that match the right side of some production.

When it finds such a substring, it *reduces* it, i.e., substitutes the left side non-terminal for the matching right side. The goal is to reduce all the way up to the start symbol and report a successful parse.

3.1.1 SHIFT REDUECE PARSING

- Shift reduce parsing attempts to construct a parse tree for an input string beginning at the leaves (bottom) and working up towards the root (top).

- This can be considered as the process of "reducing" a string w to the start symbol of a grammar.

- At each reduction step parser searches for substrings of the working string that match the right side of some production.

- When it finds such a substring, it reduces it, i.e., substitutes the left side non-terminal for the matching right side. The goal is to reduce all the way up to the start symbol and report a successful parse.

- In Shift-reduce parsing 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.

- We use $ 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:

<div align="center">

STACK   INPUT

$ w $

</div>

- During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack.

- It then reduces β to the head of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty as follows:

<div align="center">

STACK   INPUT

$ S $

</div>

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

- There are actually four possible actions a shift-reduce parser can make:

  - **Shift**
  - **Reduce**
  - **Accept**
  - **Error**

1. **Shift:** The next input symbol is shifted onto the top of the stack.
2. **Reduce:** The parser knows the right end of the string to be reduced must be at the top of the stack. It must then locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. **Accept:** Announce successful completion of parsing.
4. **Error:** Discover a syntax error has occurred and calls an error recovery routine.

EXAMPLE

Following figure steps through the actions a shift-reduce parser might take in parsing the input string $id_1 * id_2$ according to the expression grammar.

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

STACK	INPUT	ACTION
$\$$	$id_1 * id_2 \$$	shift
$\$ id_1$	$* id_2 \$$	reduce by $F \rightarrow id$
$\$ F$	$* id_2 \$$	reduce by $T \rightarrow F$
$\$ T$	$* id_2 \$$	shift
$\$ T *$	$id_2 \$$	shift
$\$ T * id_2$	$\$$	reduce by $F \rightarrow id$
$\$ T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$ T$	$\$$	reduce by $E \rightarrow T$
$\$ E$	$\$$	accept

*Configurations of a shift-reduce parser on input idl*id2*

3.1.1.1 OPERATOR PRECEDENCE PARSING

- Bottom-up parsers for a large class of context-free grammars can be easily developed using operator grammars.

- In an operator grammar, no production rule can have:
    - at the right side (no production).
    - two adjacent non-terminals at the right side.

- This property enables the implementation of efficient *operator-precedence parsers*.

EXAMPLE

E→AB	E→EOE	E→E+E |
A→a	E→id	E*E |
B→b	O→+|*|/	E/E | id

not operator grammar	not operator grammar	operator grammar

Precedence Relations

- In operator-precedence parsing, we define three precedence relations between certain pairs of terminals as follows:

Relation	Meaning
$a <\cdot b$	a yields precedence to b *(b has higher precedence than a)*
$a =\cdot b$	a has the same precedence as b
$a \cdot> b$	a takes precedence over b *(b has lower precedence than a)*

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

    <. with marking the left end,

    $=\cdot$ appearing in the interior of the handle, and

    .> marking the right end.

- In our input string \$a1a2...an\$, we insert the precedence relation between the pairs of terminals.

- Example: Consider the string id + id * id and the grammar is:

$$E \to E+E \mid E\text{-}E \mid E*E \mid E/E \mid (E) \mid \text{-}E \mid id$$

➕ The corresponding precedence relations is

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

➕ Then the string with the precedence relations inserted is:

$ <. id .> + <. id .> * <. id .> $

➕ <. is inserted between the leftmost $ and id since <. is the entry in row $ and column id.

Handle And Handle Pruning

➕ In our input string $a1a2...an$, we insert the precedence relation between the pairs of terminals.

➕ Example: Consider the string id + id * id and the grammar is:

➕ Handle is a substring that matches with the right side of the production and if the substring matches with the right side of the production, it is reduced with the non-terminal on the left side of the production

EXAMPLE

Consider the grammar

E → E+E

E → E*E

E → (E)

E→ id

And the input string is $id_1 + id_2 * id_3$

The rightmost derivation is:

E → E + E

➔ E + E * E
➔ E + E * id_3
➔ E + id_2 * id_3
➔ Id_1 + id_2 * id_3

In the above derivation, the underlined substrings are called handles.

HANDLE PRUNING

The process of obtaining rightmost derivation in reverse order is called "handle pruning". (i.e.) if w is a sentence or string of the grammar at hand, then w = γ_n where γ_n is the nth right sentinel form of some rightmost derivation.

3.1.2 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; the "L" is for left to right scanning of input symbol, the "R" for constructing right most derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decision.

There are three types of LR parsing,

> ➤ SLR (Simple LR)
> ➤ CLR (Canonical LR)
> ➤ LALR (Lookahead LR)

The schematic form of LR parser is given below.

Model of an LR Parser
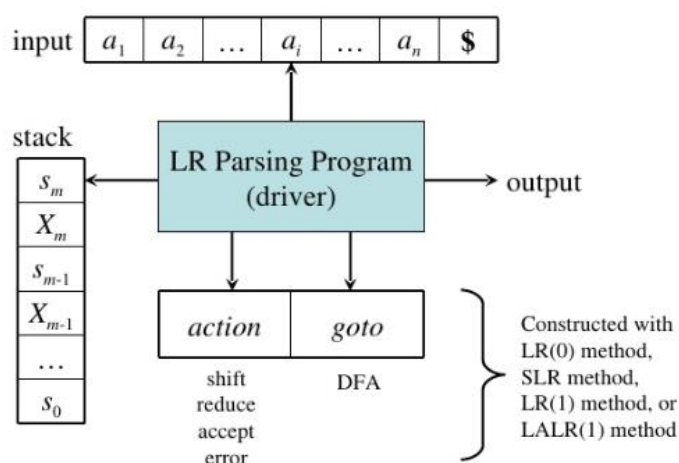
Requirements of LR parser:

> ➤ Input Buffer
> ➤ Stack

➢ Parsing Table
➢ LR Parsing Program

Input Buffer

The parsing program reads characters from an input buffer one at a time.

Stack

- Parsing program uses a stack to store string of the form $s_0X_1s_1X_2s_2X_3 \ldots\ldots\ldots X_ms_m$ where s_m is on the top.

- X_i is a grammar symbol and S_i is the state

- Each state symbol summarizes the information contained in the stack below it

- 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

Parsing Table

Consist of 2 parts
1. parsing action function action
2. goto function goto

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:

    i. Shift *j*, where *j* is a state.
    ii. Reduce by a grammar production A---> β.
    iii. Accept.
    iv. Error.

Goto
This function takes a state and grammar symbol as arguments and produces a state. If GOTO [Ii ,A] = Ij, the GOTO also maps a state *i* and nonterminal A to state *j*.

Parsing Program works as follows

- It determines s_m, the state currently on top of stack and a_i the current input symbol

- It consults parsing action table entry for action[s_m,a_i] which can have 4 values:

- The function goto takes a state and grammar symbol as arguments and produces a state.

    1. **Shift s, where s is a state**
    2. **Reduce by a grammar production A→b**
    3. **Accept**
    4. **Error**

- The function goto takes a state and grammar symbol as arguments and produces a state.

- A configuration of LR parser is a pair whose first component is stack contents and second component is remaining input:

$$(s_0 \ X_1 \ s_1 \ X_2 \ . \ . \ . X_m s_m, \ a_i \ a_{i+1} \ . \ . \ . \ a_n \ \$)$$

- The next move of the parser is determined by reading ai, the current input symbol and sm, state on stack top and then consulting parsing action table entry action[sm,ai].

- The configuration resulting after each of four types of moves are as follows:

1. If action $[s_m, a_i]$ = shift s, the parser executes a shift move, entering the configuration

$$(s_0 \ X_1 \ s_1 \ X_2 \ . \ . \ . X_m s_m, \ a_i \ a_{i+1} \ . \ . \ . \ a_n \ \$)$$

Here the parser has shifted both the current input symbol a_i and the next state s, which is given in action $[s_m, a_i]$, onto the slack; a_{i+1} becomes the current input symbol.

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

$$(s_0 \ X_1 \ s_1 \ X_2 \ s_2 \ . \ . \ . \ X_{m-r} \ s_{m-r} \ A \ s, \ a_i \ a_{i+1} \ . \ . \ . \ a_n \ \$)$$

where s = *goto* $[s_{m-r} , A]$ and r is the length of β , the right side of the production. Here the parser first popped 2r symbols off the stack (r state symbols and r grammar symbols), exposing state s_{m-r}. The parser then pushed both A, the left side of the production, and 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 calls an error recovery routine.

LR Parsing Algorithm

INPUT : Input string w, LR-Parsing table with functions ACTION and GOTO for a 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 its stack, where s_0 is the initial state, and w$ in the input buffer. The parser then executes the following program until an accept or error action is encountered.

set *ip* to point to the first symbol of w$

repeat forever begin

    Iet s be the state on top of the slack and

        a the symbol pointed to by *ip*;

    if *action* [s, a]= shift s' then begin

        push a then s' on lop of the stack;

        advance *ip* to the next input symbol

    end

    else if *action* [s,a] = reduce A➔β then begin

        pop 2*|β| symbols off the stack ;

        let s' be the state now on top *of* the stack;

        push A then *goto* [s', A] on top of the stack;

        output *the* production A➔β

    end

    else if *action* [s,a] = accept then

        return

    else error();

end

Different types of LR Parsers

All LR parsers use the same parsing algorithm but the difference is only in terms of the construction of parsing tables. There are three widely used algorithms available for constructing an LR parsing Tables. Different types of LR parsers are:

- **SLR(l) - Simple LR**

  - ➢ Works on smallest class of grammar.
  - ➢ Few number of states, hence very small table.
  - ➢ Simple and fast construction.

- **LR(1) - LR parser**

  - ➢ Also called as Canonical LR parser.
  - ➢ Works on complete set of LR(1) Grammar.

➢ Generates large table and large number of states.

➢ Slow construction.

⁺ LALR(1) - Look ahead LR parser

➢ Works on intermediate size of grammar.

➢ Number of states are same as in SLR(1).

Reasons for attractiveness of LR parser

⁺ LR parsers can handle a large class of context-free grammars.

⁺ The LR parsing method is a most general non-back tracking shift-reduce parsing method.

⁺ An LR parser can detect the syntax errors as soon as they can occur.

⁺ LR grammars can describe more languages than LL grammars.

Drawbacks of LR parsers

⁺ It is too much work to construct LR parser by hand. It needs an automated parser generator.

⁺ If the grammar contains ambiguities or other constructs then it is difficult to parse in a left-to-right scan of the input.

Construction of LR Parsing Table

3 techniques for constructing LR parsing table for a grammar :-

1. **SLR**
2. **CLR**
3. **LALR**

Simple LR (SLR)

Easy to implement

Least powerful

Canonical LR

Most powerful

Most expensive

LookAhead LR(LALR)

Intermediate in power and cost between other 2

Limitations Of SLR Parser

For an SLR parser, if we are in a state with the item A -> α ., then we reduce by A -> α iff the next input symbol 'a' belongs to Follow(A).

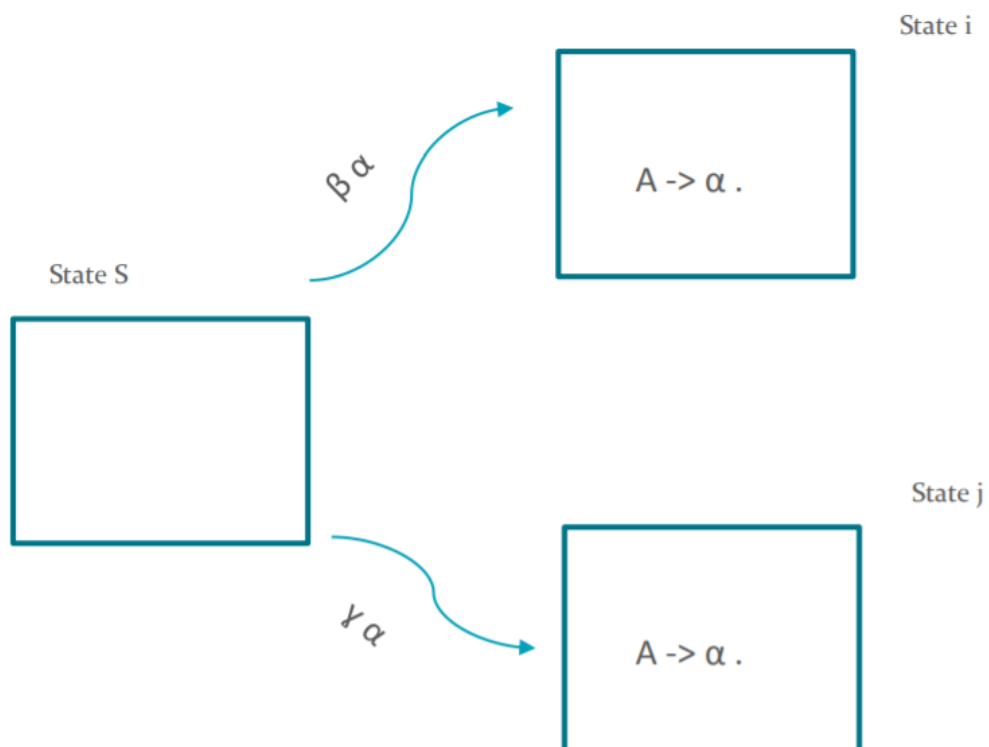This is actually a weak rule and can lead to erroneous results and/or shift/reduce conflicts.

EXAMPLE

Consider A -> α

State i: A -> α .

State j: A -> α .

String 1: γ α a

String 2: β α a

State i

A -> α .

β α

State S

State j

γ α

A -> α .

3.1.2.1 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.

- 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.

Ex: for the productions A → aBb

Possible LR(0) Items are:

> **A → .aBb**
>
> **A → a.Bb**
>
> **A → aB.b**
>
> **A → aBb.**

- 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.

- To construct the canonical LR(0) collection for a grammar we define:

  - ➢ **Augmented grammar**

  - ➢ **Two functions: CLOSURE and GOTO**

Augmented grammar

- G′ is the augmented grammar of G with a new production rule S′→S where S′ is the new starting symbol i.e,

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

- The start state of G′ = S′.

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

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.

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 $A \rightarrow \alpha .B\beta$ 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).

Computation of Closure

function closure (I)

    begin

    J := I;

      repeat

      for each item $A \rightarrow \alpha .B\beta$ in J and each production

        $B \rightarrow \gamma$ of G such that $B \rightarrow .\gamma$ is not in J do

          add $B \rightarrow .\gamma$ to J

      until no more items can be added to J

     return J

    end

The Closure Operation -- Example

$E' \rightarrow E$	closure($\{E' \rightarrow .E\}$) =
$E \rightarrow E+T$	$\{\ E' \rightarrow .E$ ⟵ kernel items
$E \rightarrow T$	$E \rightarrow .E+T$
$T \rightarrow T*F$	$E \rightarrow .T$
$T \rightarrow F$	$T \rightarrow .T*F$
$F \rightarrow (E)$	$T \rightarrow .F$
$F \rightarrow id$	$F \rightarrow .(E)$
	$F \rightarrow .id\ \}$

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 $\rightarrow \alpha$. Xβ })** 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 γ**X.**

EXAMPLE

I ={ E′ → E., E → E.+T}

goto(I,+) = { E → E+.T

T →.T*F

T →.F

F →.(E)

F →.id }

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′.

ALGORITHM

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 set of LR(0) items can be added to C.

\end

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

I₀:

E′ → .E

E → .E+T

E → .T

T → .T*F

T → .F

F → .(E)

F → .id

I₁: goto(I₀, E)

E′ → E.

E → E.+T

I₂ :goto(I₀, T)

E → T.

T → T.*F

I₃: goto(I₀, F)

T → F.

I₄ : goto(I₀, ( )

F → (.E)

E → .E+T

E → .T

T → .T*F

T → .F

F → .(E)

F → .id

I₅: goto(I₀, id)

F → id.

I₆: goto(I₁, +)

E → E+.T

T → .T*F

T → .F

F → .(E)

F → .id

I₇: goto(I₂ , *)

T → T*.F

F → .(E)

F → .id

I₈: goto(I₄ , E)

F → (E.)

E → E.+T

I₂ :goto(I₄, T)

E → T.

T → T.*F

I₃: goto(I₄, F)

T → F.

I₄ : goto(I₀, ( )

F → (.E)

E → .E+T

E → .T

T → .T*F

T → .F

F → .(E)

F → .id

I₅: goto(I₄, id)

F → id.

I₉:goto(I₆, T)

E → E+T.

T → T.*F

I₃: goto(I₆, F)

T → F.

I₄ : goto(I₆, ( )

F → (.E)

E → .E+T

E → .T

T → .T*F

T → .F

F → .(E)

F → .id

I₅: goto(I₆, id)

F → id.

I₁₀: goto(I₇, F)

T → T*F.

I₅: goto(I₇, id)

F → id.

I₄ : goto(I₇, ( )

F → (.E)

E → .E+T

E → .T

T → .T*F

T → .F

F → .(E)

F → .id

I₁₁ : goto(I₈ ,))

F → (E).

I₆: goto(I₈, +)

E → E+.T

T → .T*F

T → .F

F → .(E)

F → .id

I₇: goto(I₉, *)

T → T*.F

F → .(E)

F → .id

I0
E'→ .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I1
E'→ E.
E → E.+T
$↓
accept

I6
E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id

I9
E → E+T.
T → T.*F

I2
E → T.
T → T.*F

I7
T → T*.F
F → .(E)
F → .id

I10
T → T*F.

I5
F → id.

I4
F → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I8
E → E.+T
F → (E.)

I11
F → (E).

I3
T → F.

Constructing SLR Parsing Table

INPUT: An augmented grammar G'.

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

METHOD:

1. Construct C ={$I_0, I_1,…..I_n$}, 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→α·aβ] 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→α·] is in I_i, then set *action*[i,a] to "reduce A→α" for all a in FOLLOW(A).
    (c) If [S'→S.] is in I_i, then set *action*[i,$] to "accept".

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].

Parsing Tables of Expression Grammar

	Action Table							Goto Table		
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				

➕ Blank entries are error entries.

Number the given Grammar

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

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

|		
I₁: E′ → E.	(1, $)	====> ACCEPT
I₂ : E → T.	(2, $), (2,)), (2, +)	====> r2
I₃: T → F.	(3, $), (3, +), (3,)), (3, *)	====> r4
I₅: F → id.	(5, *), (5, +), (5,)), (5, $)	====> r6
I₉: E → E+T.	(9, $), (9,)), (9, +)	====> r1
I₁₀: T → T*F.	(10, $), (10, +), (10,)), (10, *) ====> r3
I₁₁: F → (E).	(11, *), (11, +), (11,)), (11, $) ====> r5

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.

Stack Implementation

- Check whether the given input is valid or not?

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

EXAMPLE

SLR (1) GRAMMAR

S → E

E → E + T | T

T → T * F | F

F → id

AUGMENTED GRAMMAR

S′ → S

S → E

E → E + T | T

T → T * F | F

F → id

CANONICAL COLLECTION OF LR(0) ITEMS

I_0

S′ → S

S → •E

E → •E + T

E → •T

T → •T * F

T → •F

F → •id

I_1= Go to (I_0, E)

S` → E•

E → E• + T

I_2= Go to (I_0, T)

E → T•

T →T• * F

I_3= Go to (I_0, F)

T → F•

I_4= Go to (I_0, id)

F → id•

I_5= Go to (I_1 , +)

E → E +•T

T → •T * F

T → •F

F → •id

I6= Go to (I2, *)

T → T * •F

F → •id

I_7= Go to (I_5, T)

E → E + T•
T →T• * F

I_3 = Go to (I_5, F)

T → F•

I_4= Go to (I_5, id)

F → id•

I_8= Go to (I_6, F)

T → T * F•

I4= Go to (I6, id)

F → id•

I6 = Go to (I7 , *)

T → T * •F

F → •id

<u>DRAW CANONICAL COLLECTION OF LR(0) ITEMS</u>
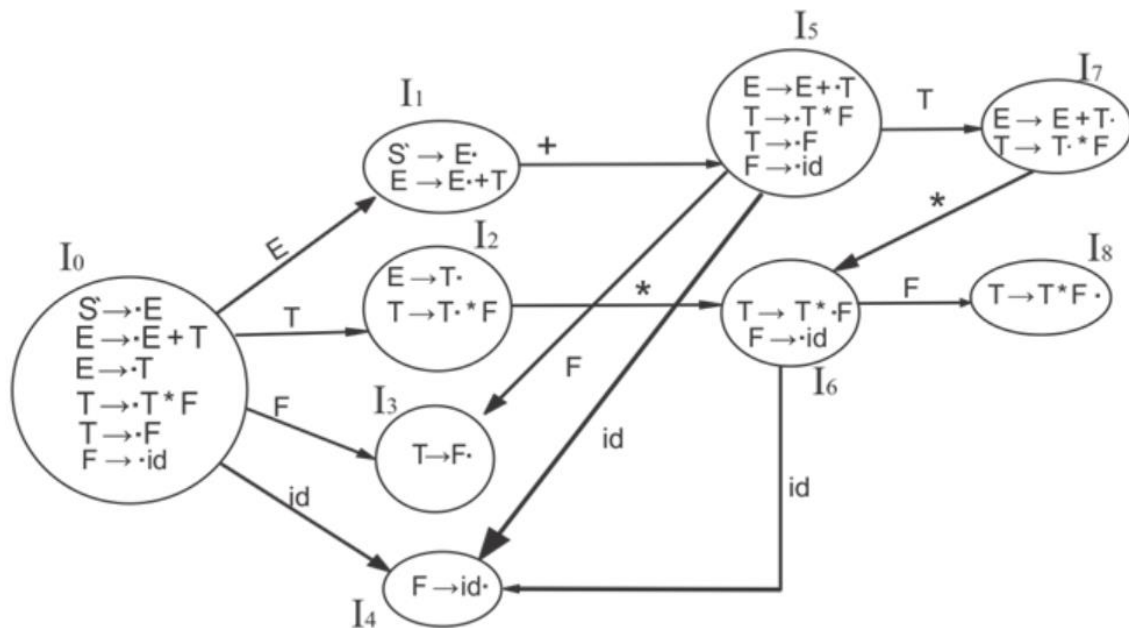
<u>NUMBER THE PRODUCTIONS</u>

1. **S → E**
2. **E → E + T | T**
3. **T → T * F | F**
4. **F → id**

<u>SLR(1) TABLE</u>

FOLLOW (S) = {$}

FOLLOW (E)= {+, $}

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

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

➤ I_1 contains the final item which drives S → E• and follow (S) = {$}, so action {$I_1$, $} = Accept

➤ I_2 contains the final item which drives E → T• and follow (E) = {+, $}, so action {$I_2$, +} = R2, action {$I_2$, $} = R2

➤ I_3 contains the final item which drives T → F• and follow (T) = {+, *, $}, so action {$I_3$, +} = R4, action {$I_3$, *} = R4, action {$I_3$, $} = R4

➢ I$_4$ contains the final item which drives F → id• and follow (F) = {+, *, $}, so action {I$_4$, +} = R5, action {I$_4$, *} = R5, action {I$_4$, $} = R5

➢ I$_7$ contains the final item which drives E → E + T• and follow (E) = {+, $}, so action {I$_7$, +} = R1, action {I$_7$, $} = R1

➢ I8 contains the final item which drives T → T * F• and follow (T) = {+, *, $}, so action {I$_8$, +} = R3, action {I$_8$, *} = R3, action {I$_8$, $} = R3.

States	Action				Go to		
	id	+	*	$	E	T	F
I$_0$	S$_4$				1	2	3
I$_1$		S$_5$		Accept			
I$_2$		R$_2$	S$_6$	R2			
I$_3$		R$_4$	R$_4$	R4			
I$_4$		R$_5$	R5	R5			
I$_5$	S4					7	3
I$_6$	S4						8
I$_7$		R1	S6	R1			
I$_8$		R3	R3	R3			

EXAMPLE

Every SLR (1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1). Consider the grammar with productions

S → L = R

S → R

L → * R

L → id

R → L

AUGMENTED GRAMMAR

S′ → S

S → L = R

S → R

L → * R

L → id

R → L

CANONICAL COLLECTION OF LR(0) ITEMS

I_0

S′ → • S

S → •L = R

S → •R

L → •* R

L → •id

R → • L

I_1= Go to (I_0, S)

S′ → S •

I_2= Go to (I_0, L)

S → L •= R

R → L•

I_3= Go to (I_0, R)

S → R•

I_4= Go to (I_0, *)

L → * •R

R → • L

L → •* R

L → •id

I_5= Go to (I_0, id)

L → id•

I_6= Go to (I_2, =)

S → L = •R

R → • L

L → •* R

L → •id

I_7= Go to (I_4, R)

L → * R•

I_8= Go to (I_4, L)

R → L•

I_4= Go to (I_4, *)

L → * •R

R → • L

L → •* R

L → •id

I_5= Go to (I_4, id)

L → id•

I_9= Go to (I_6, R)

S → L = R•

I_8= Go to (I_6, L)

R → L •

I_4= Go to (I_6, *)

L → *• R

R → • L

L → •* R

L → •id

I_5= Go to (I_6, id)

L → id•

NUMBER THE PRODUCTIONS

1. S → L = R
2. S → R
3. L → * R
4. L→ id
5. R → L

PARSING TABLE

I₁= S′ → S •

I₂= R → L•   FOLLOW (S)={$}

I₃= S → R•   FOLLOW (L)= {=,$}

I₅= L → id•   FOLLOW (R)= {=,$}

I₇= L → * R•

I₈=   R → L•

I₉= S → L = R•

$$(2, =), (2, \$) \implies r5$$
$$(3, \$) \implies r2$$
$$(5, =), (5, \$) \implies r4$$
$$(7, =), (7, \$) \implies r3$$
$$(8, \$), (8, =) \implies r5$$
$$(9, \$) \implies r1$$

STATE	ACTION				GOTO		
	id	*****	**=**	**$**	**R**	**L**	**S**
0	s5	s4			3	2	1
1				ACCEPT			
2			s6/ r5	r5			
3				r2			
4	s5	s4			7	8	
5			r4	r4			
6	s5	s4			9	8	
7			r3	r3			
8			r5	r5			
9				r1			

Here there is a shift reduce conflict in (2,=) so grammar is not SLR(1)

3.1.2.2 CONSTRUCTING CANONICAL LR(CLR) PARSING TABLES

- In SLR method, the state i makes a reduction by A→α when the current token is **a**:

    – if the A→α.is in the I_i and **a** is FOLLOW(A)

    – In some situations, when state i appears on stack top, the viable prefix βα on the stack is such that βA cannot be followed by a in right sentential form.

- Thus the reduction by A→α would be invalid on input a.

- Because of that we go for : Canonical LR Parser

    – In this, it is possible to carry more information in the state that will allow us to avoid some of these invalid reductions

- 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 → α.β,a   where **a** is the look-head of the LR(1) item

                              (**a** is a terminal or end-marker.)

- Such an object is called LR(1) item.

    – 1 refers to the length of the second component

    – The lookahead has no effect in an item of the form [A → α.β,a], where β is not ∈.

    – But an item of the form [A → α.,a] calls for a reduction by A → α only if the next input symbol is a.

- When β (in the LR(1) item A → α.β,a) is not empty, the look-head does not have any affect.

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

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.

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).

closure(I) is: (where I is a set of LR(1) items)

  - every LR(1) item in I is in closure(I)

  - if $A \rightarrow \alpha.B\beta,a$ 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) .

Construction of CLR (canonical LR) 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 G′.

METHOD:

1. **Construct C′ = { I_0, I_1, ..., 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 [$A \rightarrow \alpha.a\beta$,b] is in I_i, and GOTO (I_i, a) = I_j, then set ACTION [i, a] To "shift j". Here a must be a terminal.**
   (b) **If [$A \rightarrow \alpha.$, a] is in I_i, $A \neq S′$, then set ACTION[i, a] to "reduce $A \rightarrow \alpha$."**
   (c) **If [$S′ \rightarrow S.$, $] is in I_i, then set ACTION[i, $] to "accept".**

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

3. **The goto transitions for state i are constructed for all non terminals 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′ $\rightarrow$ .S, $].**

EXAMPLE

Construct CLR parsing table for the flowing grammar

S→ CC

C→ cC | d

AUGMENTED GRAMMAR

S'→ S

S→ CC

C→ cC

C→ d

CANONICAL COLLECTION OF LR(1) ITEMS

I₀

S' → •S , $

S→ •CC, $

C→ •cC, c | d

C→ •d, c | d

I₁ = GOTO (I₀, S)

 S' → S•, $

I₂ = GOTO (I₀, C)

S → C•C, $

C → •cC, $

C → •d, $

I₃ = GOTO (I₀, c)

C → c•C, c | d

C → •c C , c | d

C → •d, c | d

I₄ = GOTO (I₀, d)

C → d• , c | d

I₅ = GOTO (I₂, C)

S → CC•, $

I₆ = GOTO (I₂, c)

C → c•C, $

C → •c C ,$

C → •d , $

I₇ = GOTO (I₂, d)

C → d• , $

I₈ = GOTO (I₃, C)

C → c C•, c | d

I₃ = GOTO (I₃, c)

C → c• C , c | d

C→ •cC, c | d

C→ •d, c | d

I₄ = GOTO (I₃, d)

C→ d •, c | d

I₉ = GOTO (I₆, C)

C → c C• , $

I₆ = GOTO (I₆, c)

C → c• C ,$

C→ •cC, $

C→ •d, $

I₇ = GOTO (I₆, d)

C→ d•, $

GOTO GRAPH FOR THE GRAMMAR
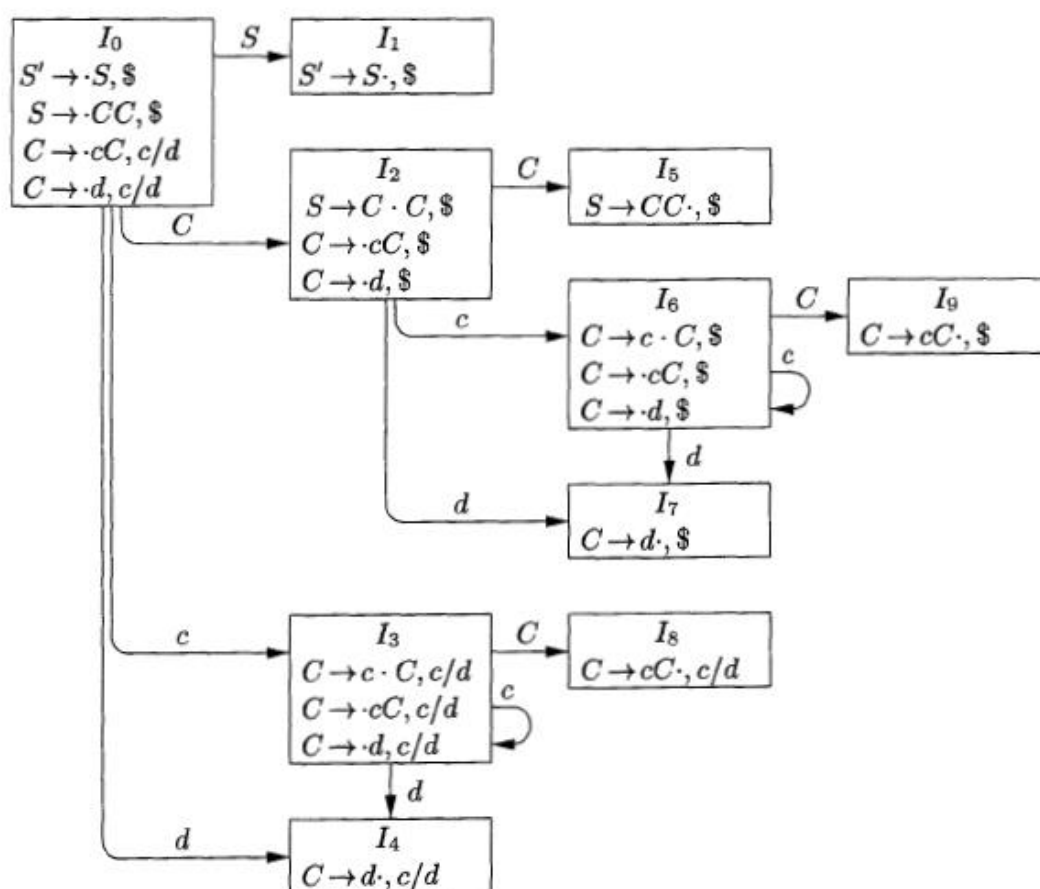
NUMBER THE PRODUCTIONS

1. S→ CC
2. C→ cC
3. C→ d

PARSING TABLE

I₁ = $S' \rightarrow S\bullet$, $\$

I₄ = $C \rightarrow d\bullet$, c | d

I₅ = $S \rightarrow CC\bullet$, $\$

I₇ = $C \rightarrow d\bullet$, $\$

I₈ = $C \rightarrow c\ C\bullet$, c | d

I₉ = $C \rightarrow c\ C\bullet$, $\$

STATE	ACTION			GOTO	
	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		

27

3.1.2.3 CONSTRUCTING LALR PARSING TABLE

- **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.

Creating LALR Parsing Tables

- Canonical LR(1) Parser ➔ LALR Parser

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

$$Ex: \quad S \rightarrow L.=R,\$ \quad ➔ \quad S \rightarrow L.=R \qquad Core$$

$$R \rightarrow L.,\$ \qquad R \rightarrow L.$$

- 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.

$$I_1:L \rightarrow id.,= \qquad\qquad A \text{ new state:} \quad I_{12}: L \rightarrow id.,=$$

$$➔ \qquad\qquad L \rightarrow id.,\$$$

$$I_2:L \rightarrow id.,\$ \qquad \text{have same core, merge them}$$

- 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.

We need to do following update in the parsing table.

- Consider a pair of similar looking states (same kernel and different lookaheads) in the set of LR(1) items
  I_4: C → d. , c/d I_7: C → d., $

- Replace I_4 and I_7 by a new state I_{47} consisting of (C → d., c/d/$)

- Similarly I_3 & I_6 and I_8 & I_9 form pairs

- Merge LR(1) items having the same core

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 ∪ I_2 ∪ ... ∪ 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.

STATE	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		

****