Marwadi University
Marwadi Chandarana Group
NAAC A+

CD : COMPILER DESIGN
Parsing

Department of CE

Unit no : 3
Parsing
(01CE0714)

Prof. Shilpa Singhal

Outline :

Role of parser

Parse tree

Classification of grammar

Derivation and Reduction

Ambiguous grammar

Left Recursion

Left Factoring

Top-down Bottom-up parsing

LR Parsers – LR(0), SLR, CLR , LALR

Department of CE

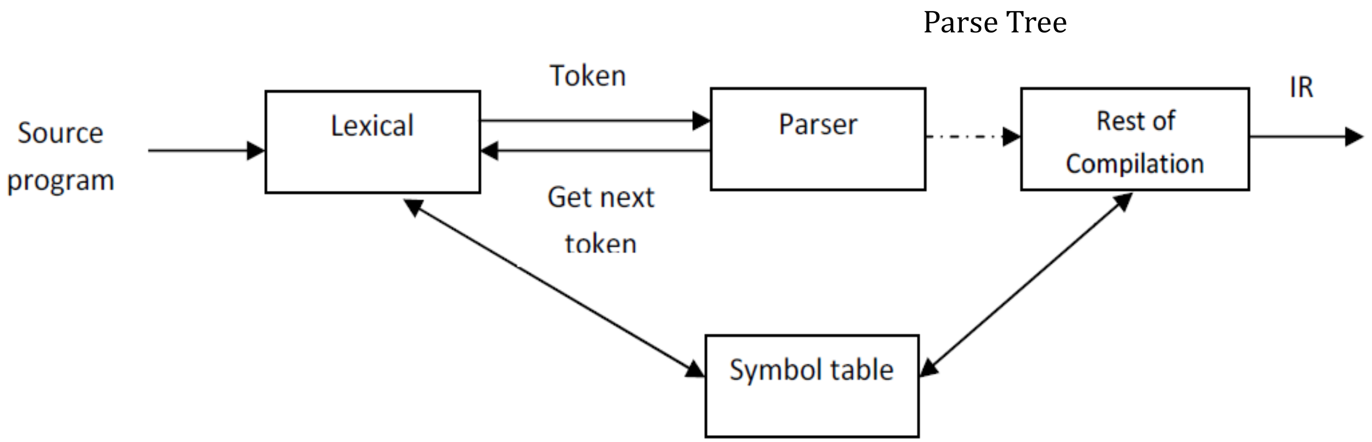
Unit no : 3
Parsing
(01CE0714)

Prof. Shilpa Singhal

Role of Parser

- In our compiler model, the parser obtains a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source language.
- It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

Scanner – Parser Interaction



- For well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

Syntax Error Handling

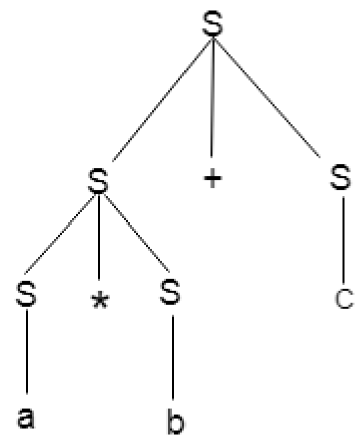
- If a compiler had to process only correct programs, its design and implementation would be greatly simplified.
- But programmers frequently write incorrect programs, and a good compiler should assist the programmer in identifying and locating errors.
- We know that programs can contain errors at many different levels. For example, errors can be
- **Lexical** : Such a misspelling an identifier, keyword, or operator
- **Syntactic** : Such as arithmetic expression with unbalanced parenthesis
- **Semantic** : Such as an operator applied to incompatible operand
- **Logical** : Such as infinitely recursive call

Syntax Error Handling

- The error handler in a parser has simple-to-state goals :
- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect sub sequent errors.
- It should not significantly slow down the processing of correct programs.

Parse tree

- Parse tree is graphical representation of symbol. Symbol can be terminal as well as non-terminal.
- The root of parse tree is start symbol of the string.
- Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.
- Example:-



Classification of Grammar

- Grammars are classified on the basis of production they use (Chomsky, 1963).
- Given below are class of grammar where each class has its own characteristics and limitations.

1. **Type-0 Grammar:- Recursively Enumerable Grammar**
   - These grammars are known as phrase structure grammars. Their productions are of the form,
   - $\alpha = \beta$, where both α and β are terminal and non-terminal symbols.
   - This type of grammar is not relevant to Specifications of programming languages.

2. **Type-1 Grammar:- Context Sensitive Grammar**
   - These Grammars have rules of the form $\alpha A\beta \rightarrow \alpha \Upsilon \beta$ with A nonterminal and α, β, Υ strings of terminal and nonterminal symbols. The string α and β may be empty but Υ must be nonempty.
   - Eg:- AB->CDB

     Ab->Cdb

     A->b

Classification of Grammar

3. **Type-2 Grammar:- Context Free Grammar**
   - These are defined by the rules of the form A → ϒ, with A nonterminal and ϒ a sting of terminal and nonterminal Symbols. These grammar can be applied independent of its context so it is Context free Grammar (CFG). CFGs are ideally suited for programming language specification.
   - Eg:- A → aBc

4. **Type-3 Grammar:- Regular Grammar**
   - It restrict its rule of single nonterminal on the left hand side and a right-hand side consisting of a single terminal, possibly followed by a single nonterminal. The rule S → ϵ is also allowed if S does not appear on the right side of any rule.
   - Eg:- A → ϵ

     A → a

     A → aB

Derivation

- Let production P_1 of grammar G be of the form

  $P_1 : A ::= \alpha$

  and let β be a string such that $\beta = \gamma A \theta$, then replacement of A by α in string β constitutes a derivation according to production P_1.

- Example

  <Sentence> ::= <Noun Phrase><Verb Phrase>

  <Noun Phrase> ::= <Article> <Noun>

  <Verb Phrase> ::= <Verb><Noun Phrase>

  <Article> ::= a | an | the

  <Noun> ::= boy | apple

  <Verb> ::= ate

Derivation

- The following strings are ***<u>sentential form</u>***.

  <Sentence>

  <Noun Phrase> <Verb Phrase>

  the boy <Verb Phrase>

  the boy <verb> <Noun Phrase>

  the boy ate <Noun Phrase>

  the boy ate an apple

Derivation

- The process of deriving string is called Derivation and graphical representation of derivation is called derivation tree or parse tree.
- Derivation is a sequence of a production rules, to get the input string.
- During parsing we take two decisions:

1) Deciding the non terminal which is to be replaced.
2) Deciding the production rule by which non terminal will be replaced.

For this we are having:

1) Left most derivation

2) Right most derivation

Left Derivation

- A derivation of a string S in a grammar G is a left most derivation if at **every step the left most non terminal is replaced.**

Example:

- Production:

S→ S + S

S→ S * S

S→ id

- String:- id+id*id

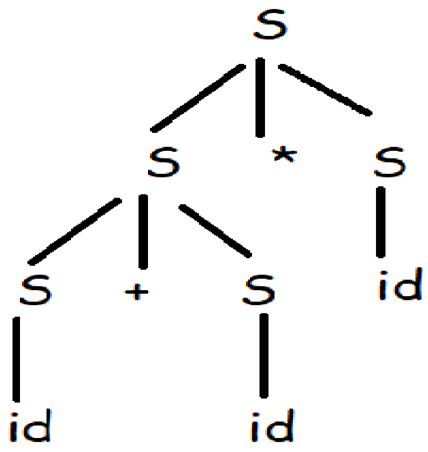
S→ S * S

S→S + S * S

S→ id + S * S

S→ id + id * S

S→ id + id * id



Right Derivation

- A derivation of a string S in a grammar G is a right most derivation if **at every step the Right most non terminal is replaced.**

Example:

- Production:

S→ S + S

S→ S * S

S→ id

- String:- id+id*id

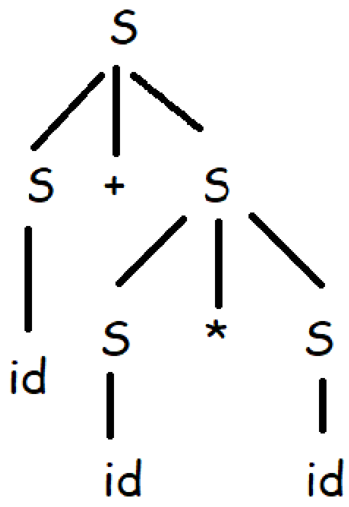
S→ S + S

S→S + S * S

S→ S + S * id

S→ S + id * id

S→ id + id * id



Left Derivation and Right Derivation

Derive the string "abb" for leftmost derivation and rightmost derivation using a CFG given by,

$S \rightarrow AB \mid \varepsilon$

$A \rightarrow aB$

$B \rightarrow Sb$

Leftmost derivation:

S

AB

$\boxed{aB}\ B$

$a\ \boxed{Sb}\ B$

$a\ \boxed{\varepsilon}\ bB$

$ab\ \boxed{Sb}$

$ab\ \boxed{\varepsilon}\ b$

abb

Rightmost derivation:

S

AB

$A\ \boxed{Sb}$

$A\ \boxed{\varepsilon}\ b$

$\boxed{aB}\ b$

$a\ \boxed{Sb}\ b$

$a\ \boxed{\varepsilon}\ bb$

abb

Left Derivation and Right Derivation

1.Derive the string "aabbabba" for leftmost derivation and rightmost derivation using a CFG given by,

$S \rightarrow aB \mid bA$

$A \rightarrow a \mid aS \mid bAA$

$B \rightarrow b \mid bS \mid aBB$

2.Derive the string "00101" for leftmost derivation and rightmost derivation using a CFG given by,

$S \rightarrow A1B$

$A \rightarrow 0A \mid \varepsilon$

$B \rightarrow 0B \mid 1B \mid \varepsilon$

Left Derivation and Right Derivation

Soution.1

Leftmost derivation:

S	
aB	$S \rightarrow aB$
aaBB	$B \rightarrow aBB$
aabB	$B \rightarrow b$
aabbS	$B \rightarrow bS$
aabbaB	$S \rightarrow aB$
aabbabS	$B \rightarrow bS$
aabbabbA	$S \rightarrow bA$
aabbabba	$A \rightarrow a$

Rightmost derivation:

S	
aB	$S \rightarrow aB$
aaBB	$B \rightarrow aBB$
aaBbS	$B \rightarrow bS$
aaBbbA	$S \rightarrow bA$
aaBbba	$A \rightarrow a$
aabSbba	$B \rightarrow bS$
aabbAbba	$S \rightarrow bA$
aabbabba	$A \rightarrow a$

Left Derivation and Right Derivation

Soution.2

Leftmost derivation:	**Rightmost derivation:**
S	S
A1B	A1B
0A1B	A10B
00A1B	A101B
001B	A101
0010B	0A101
00101B	00A101
00101	00101

Reduction

Let production P_1 of grammar G be of the form

$$P_1 : A ::= \alpha$$

and let σ be a string such that $\sigma = \gamma\ \alpha\ \theta$, then replacement of α by A in string σ constitutes a reduction according to production P_1.

Step	String
0	the boy ate an apple
1	<Article> boy ate an apple
2	<Article> <Noun> ate an apple
3	<Article> <Noun> <Verb> an apple
4	<Article> <Noun> <Verb> <Article> apple
5	<Article> <Noun> <Verb> <Article> <Noun>
6	<Noun Phrase> <Verb> <Article> <Noun>
7	<Noun Phrase> <Verb> <Noun Phrase>
8	<Noun Phrase> <Verb Phrase>
9	<Sentence>

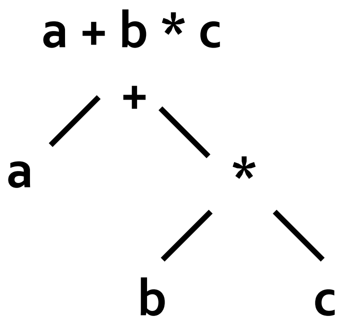
Ambiguous Grammar

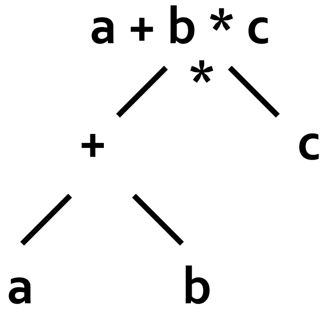
- A CFG is said to be **ambiguous** if there exists more than one derivation tree for the given input string i.e., more than one LeftMost Derivation Tree (LMDT) or RightMost Derivation Tree (RMDT).
- It implies the possibility of different interpretation of a source string.
- Existence of ambiguity at the level of the syntactic structure of a string would mean that more than one parse tree can be built for the string. So string can have more than one meaning associated with it.

Ambiguous Grammar

E → Id| E + E | E * E

Id → a | b | c



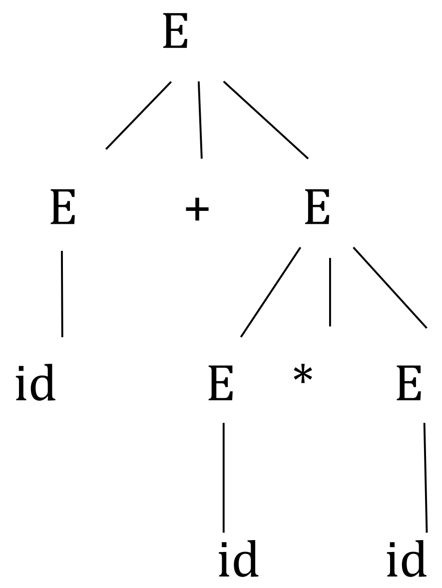


Both tree have same string : a + b * c

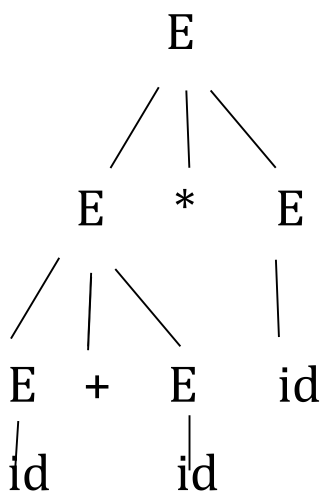
Ambiguous Grammar

E→ E + E | E * E | id

By parse tree:-



Parse tree-1



Parse tree-2

Ambiguous Grammar Example:-

Prove that given grammar is ambiguous grammar:

E→ a | Ea | bEE | EEb | EbE

Ans:-

Assume string baaab

E→ bEE

baE

baEEb

baaEb

baaab

Left derivation-1

OR

E→ EEb

bEEEb

baEEb

baaEb

baaab

Left derivation-2

Exercise: Ambiguous Grammar

Check whether following grammars are ambiguous or not:

1. $S \rightarrow aS \mid Sa \mid \epsilon$ (string: aaaa)
2. $S \rightarrow aSbS \mid bSaS \mid \epsilon$ (string: abab)
3. $S \rightarrow SS+ \mid SS^* \mid a$ (string: aa+a*)

Left Recursion

- In leftmost derivation by scanning the input from left to right, grammars of the form **$A \rightarrow A\ x$** may cause endless recursion.
- Such grammars are called **left-recursive** and they must be transformed if we want to use a top-down parser.
- Example:

$E \rightarrow Ea \mid E+b \mid c$

Algorithm

- Assign an ordering from A1,…….An to the non terminal of the grammar;
- For i = 1 to n do

begin

for j=1 to i-1 do

begin

replace each production of the form $A_i \rightarrow A_i\gamma$

by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots\ldots \mid \delta_k\gamma$

where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots\ldots\ldots\ldots \mid \delta_k$ are all current Aj production.

end

eliminate the intermediate left recursion among A_i productions.

end

Left Recursion

- There are three types of left recursion:

  direct ($A \rightarrow A\ x$)

  indirect ($A \rightarrow B\ C,\ B \rightarrow A$)

  hidden ($A \rightarrow B\ A,\ B \rightarrow \varepsilon$)

To eliminate direct left recursion replace

$$\mathbf{A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n}$$

with

$$\mathbf{A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'}$$
$$\mathbf{A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon}$$

Example

1. $E \rightarrow E + T \mid T$

   $T \rightarrow T * F \mid F$

   $F \rightarrow (E) \mid id$

Ans.

$$A \rightarrow A\alpha \mid \beta$$

Replace with,

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Example

1. $A \rightarrow Aad \mid Afg \mid b$

Ans:-

Remove left recursion

$A \rightarrow bA'$

$A' \rightarrow adA' \mid fgA' \mid \varepsilon$

2. $A \rightarrow Acd \mid Ab \mid jk$

   $B \rightarrow Bh \mid n$

Ans :-

Remove left recursion

$A \rightarrow jkA'$

$A' \rightarrow cdA' \mid bA' \mid \varepsilon$

$B \rightarrow nB'$

$B' \rightarrow hB' \mid \varepsilon$

Example

3. $E \rightarrow Aa \mid b$

   $A \rightarrow Ac \mid Ed \mid \varepsilon$

Ans:-

Replace E,

$E \rightarrow Aa \mid b$

$A \rightarrow Ac \mid \mathit{Aad} \mid \mathit{bd} \mid \varepsilon$

Remove left recursion

$E \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid A'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Left Factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
- Consider,
  - $S \rightarrow$ **if** E **then** S **else** S | **if** E **then** S
  - Which of the two productions should we use to expand non-terminal S <u>when the next token is **if**</u>?
  - We can solve this problem by factoring out the common part in these rules. This way, we are postponing the decision about which rule to choose until we have more information (namely, whether there is an **else** or not).
  - This is called **left factoring**

Algorithm

- For each non terminal A find the longest prefix **α** common to two or more of its alternative.
- If **α!=E,** i.e, there is a non trivial common prefix, replace all the A productions **A→ αβ1 | αβ2 | ….. | αβn | ϒ ,**

where ϒ represents all the alternative which do not starts with α by,

A→ αA’ | ϒ

A’→ β1 | β2 |…… | βn

Here, A’ is new non terminal, repeatedly apply this transformation until no two alternatives for a non-terminal have a common prefix.

Left Factoring

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$

becomes

$A \rightarrow \alpha\ A'' \mid \gamma$

$A'' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Left Factoring Example

E -> T+E | T

T -> V*T | V

V-> id

Ans.

E→ TE'
E' → +E | ε

T→ VT'

T' → *T | ε

V → id

Left Factoring Example

1. **S → cdLk | cdk | cd**

   L→ mn | ε

Ans.

S→ cdS'

S' → Lk | k | ε

L→ mn | ε

2. **E → iEtE | iEtEeE | a**

   A→b

Ans.

E → iEtEE' | a

E' → ε | eE

A → b

Left Factoring Example

3. A → xByA | xByAzA | a

Ans.

A→ xByAA'|a

A' → ε| zA

4. A → aAB | aA| a

Ans.

A→ aA'

A' → AB| A | ε

A' → AA''|ε

A''→ B|ε

Rules to Compute First()

1. If $A \rightarrow \alpha$ and α is terminal, add α to $FIRST(A)$.
2. If $A \rightarrow \in$, add $\in$ to $FIRST(A)$.
3. If X is nonterminal and X→$Y_1\, Y_2 \ldots . Y_k$ is a production, then place a in $FIRST(X)$ if for some i, a is in $FIRST(Yi)$, and ϵ is in all of $FIRST(Y_1), \ldots \ldots \ldots, FIRST(Y_{i-1})$; that is $Y_1 \ldots Y_{i-1} \Rightarrow \epsilon$. If ϵ is in $FIRST(Y_j)$ for all $j = 1,2, \ldots \ldots, k$ then add ϵ to $FIRST(X)$.

   Everything in $FIRST(Y_1)$ is surely in $FIRST(X)$ If Y_1 does not derive ϵ, then we do nothing more to $FIRST(X)$, but if $Y_1 \Rightarrow \epsilon$, then we add $FIRST(Y_2)$ and so on.

Compute First

Example :

E → E+T|T

T→ T*F|F

F → (E)|id

Terminals(Σ): id, if, +, -, *, /, (,), 0-9, a-z, ↑,
punctuation symbols(comma, colon, semicolon, etc….)

Step 1: Remove left recursion

E→ TE'

E'→ +TE' | ϵ

T→ FT'

T'→*FT' | ϵ

F→ (E) | id

Step 2: Perform left factoring

(No Left factoring in above grammar)

Step 3: Find First and Follow of Non terminal

NT	First
E	{ (, id }
E'	{ +, ϵ }
T	{ (, id }
T'	{ *, ϵ }
F	{ (, id }

Compute First

Example :

A → bcg | gh

FIRST(A) = {b, g}

A → Bcd | gh

B → m | ε

FIRST(A) = {m, c , g}

FIRST(B) = {m , ε}

Compute First

Example :

A → BCD | Cx

B → b | **ε**

C → c | ε

D → d | ε

FIRST(A) = {b, c, d, x, ε}

FIRST(B) = {b, ε}

FIRST(C) = {c , ε}

FIRST(D) = {d , ε}

Compute First

Example :

S → PQr | s

P → Abc | ε

Q → d | ε

A → a| ε

A → mn | Xy | Z

X → x | ε

Z → ε

Rules to compute Follow()

1. Place $ $in\ follow(S)$. (S is start symbol)
2. If A $\rightarrow \alpha B\beta$, then everything in $FIRST(\beta)$ except for ϵ is placed in $FOLLOW(B)$
3. If there is a production A $\rightarrow \alpha B$ or a production A $\rightarrow \alpha B\beta$ where $FIRST(\beta)$ contains ϵ then everything in $FOLLOW(A) = FOLLOW(B)$

Compute Follow()

E→E+T | T
T→T*F | F
F→(E) | id

Step 1: Remove left recursion

E→ TE'
E'→ +TE' | ϵ
T→ FT'
T'→*FT' | ϵ
F→ (E) | id

Step 2: Perform left factoring

(No Left factoring in above grammar)

Step 3: Find First and Follow of Non terminal

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +, $,) }
T'	{ *, ϵ }	{ +, $,) }
F	{ (,id }	{*, +, $,) }

Compute Follow

Example :

A → bcg | gh

FOLLOW(A) = {$}

A → Bcd | gh

B → mA | ε

FOLLOW(A) = {$, c}

FOLLOW(B) = {c}

Compute Follow

Example :

A → BCD | Cx

B → b | ε

C → c | ε

D → d | ε

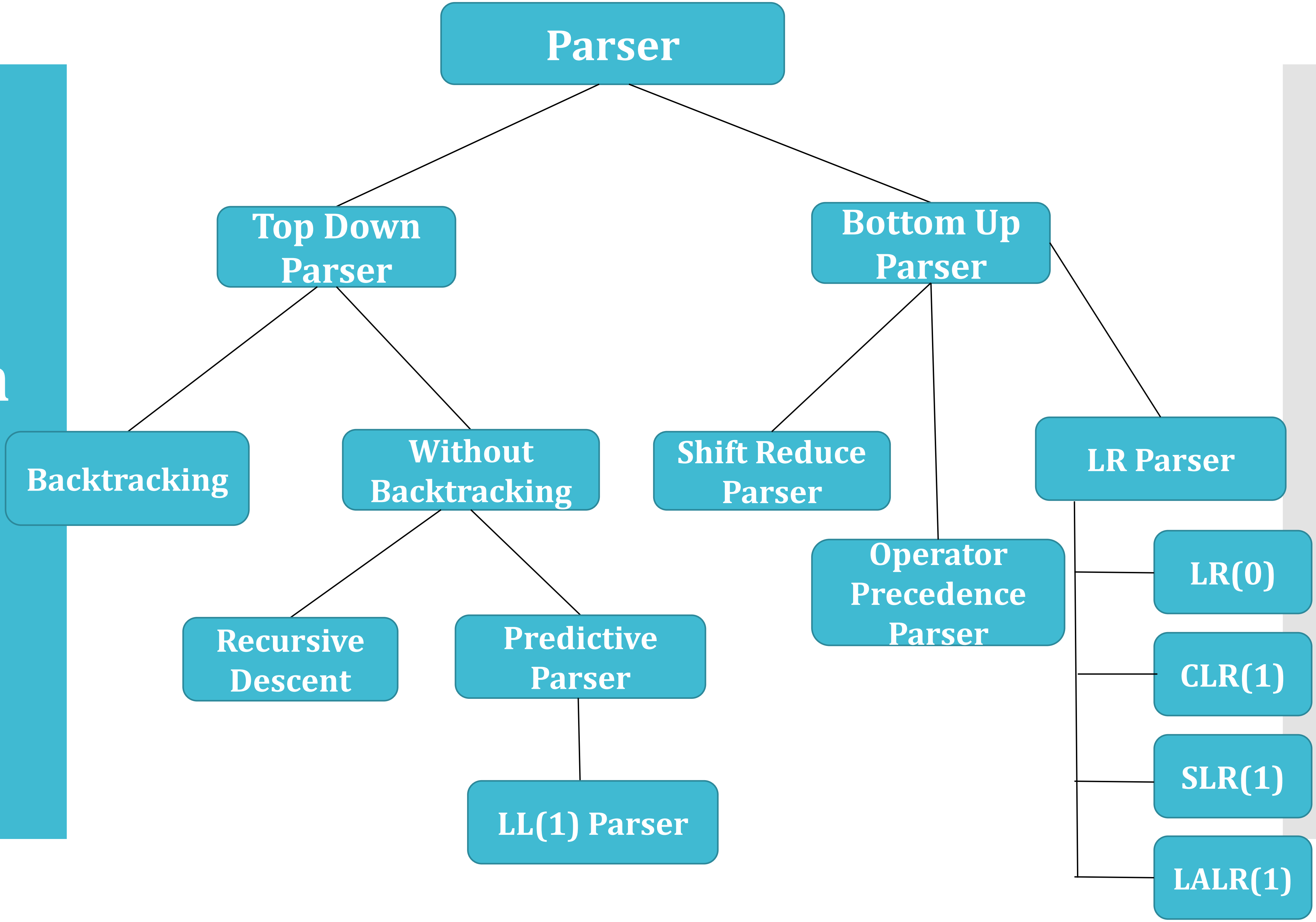
Parsing

- Parsing is a technique that takes input string and produces output either a parse tree if string is valid sentence of grammar, or an error message indicating that string is not a valid.

- Types of parsing are:

1. **Top down parsing**: In top down parsing parser build parse tree from top to bottom.
2. **Bottom up parsing**: Bottom up parser starts from leaves and work up to the root.

Classification of Parser



Top down parser

Top-down parser is the parser which generates parse tree for the given input string with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals.

It works as following:

- Start with the root or start symbol.
- Grow the tree downwards by expanding productions at the lower levels of the tree.
- Select a nonterminal and extend it by adding children corresponding to the right side of some production for the nonterminal.
- Repeat till, Lower fringe consists only terminals and the input is consumed Top-down parsing basically finds a leftmost derivation for an input string.

Limitation of Top down parser

The limitation of predictive parsing and all Top-Down parsing algorithm is that if the grammar is **Left Recursive or Left factoring** then **Top-Down parsing without backtracking** does not work.

Limitation of Top down parser

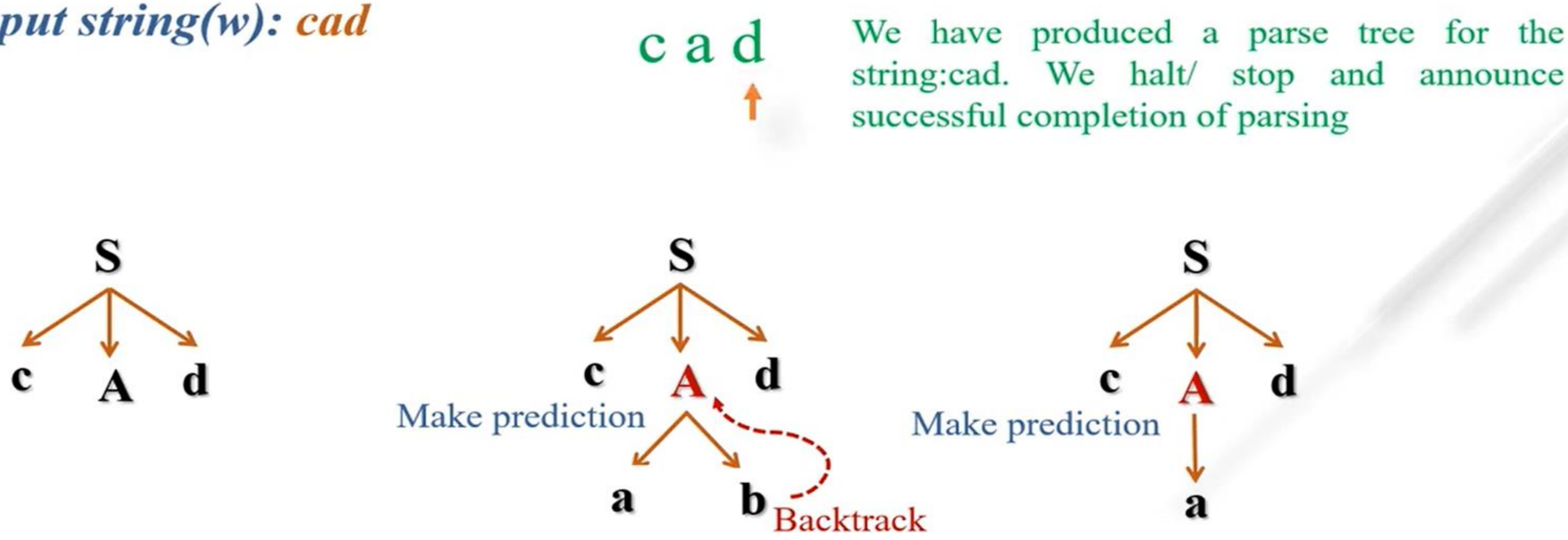
The limitation of predictive parsing and all Top-Down parsing algorithm is that if the grammar is **Left Recursive or Left factoring** then **Top-Down parsing without backtracking** does not work.

1.Backtracking

- In backtracking, expansion of nonterminal symbol we **choose one alternative** and **if any mismatch occurs** then we **try another alternative**.
- Grammar: **S→ cAd**

  A→ ab | a

  Input string(w): cad



Top down parser with backtracking

Example:

E→ T + E | T
T→ V * T | V
V→ <id>

I/p string is:
<id> + <id> * <id>

	SSM	Prediction	String
E	1	--	<id>+<id>*<id>
T+E	1	E→ T+E	<id>+<id>*<id>
V*T+E	1	T→V*T	<id>+<id>*<id>
<id>*T+E	~~1~~ 2	V→<id>	<id>+<id>*<id>
V+E	1	T→V	<id>+<id>*<id>
<id>+E	~~1~~ ~~2~~ 3	V→<id>	<id>+<id>*<id>
<id>+T+E	3	E→T+E	<id>+<id>*<id>
<id>+V*T+E	3	T→V*T	<id>+<id>*<id>
<id>+<id>*T+E	~~3~~ ~~4~~ 5	V→<id>	<id>+<id>*<id>
<id>+<id>*V*T+E	5	T→V*T	<id>+<id>*<id>
<id>+<id>*<id>*T+E	~~5~~	V→<id>	<id>+<id>*<id>
<id>+T	3	E→T	<id>+<id>*<id>
<id>+V*T	3	T→V*T	<id>+<id>*<id>
<id>+<id>*T	~~3~~ ~~4~~ 5	V→<id>	<id>+<id>*<id>
<id>+<id>*V*T	5	T→V*T	<id>+<id>*<id>
<id>+<id>*<id>*T	5	V→<id>	<id>+<id>*<id>
<id>+<id>*V	5	T→V	<id>+<id>*<id>
<id>+<id>*<id>	5	V→<id>	<id>+<id>*<id>

Top down parser without backtracking Example:

E→ T + E | T
T→ V * T | V
V→ <id>

E→ TE'
E'→ +E | ε
T → VT'
T'→ *T | ε
V→ <id>

I/p string is:
<id> + <id> * <id>

CSF	Symbol	Prediction
E	<id>	E→ TE'
TE'	<id>	T→ VT'
VT'E'	<id>	V→<id>
<id>T'E'	+	T'→ε
<id>E'	+	E'→+E
<id>+E	<id>	E→TE'
<id>+TE'	<id>	T→VT'
<id>+VT'E'	<id>	V→<id>
<id>+<id>T'E'	*	T'→*T
<id>+<id>*TE'	<id>	T→VT'
<id>+<id>*VT'E'	<id>	V→<id>
<id>+<id>*<id>T'E'	--	T'→ε
<id>+<id>*<id>E'	--	E'→ε
<id>+<id>*<id>	--	--

2.Recursive Descent Parser

- A Recursive Descent Parser is a variant of top down parsing that executes set of recursive procedure to process the input without backtracking.
- It may involve backtracking.
- There is a procedure for each non terminal in the grammar.
- RHS of the production rule is considered as the definition of procedure.
- As it reads the expected input symbol, it advances the input pointer to next position.

Recursive Descent Parser Example

$E \rightarrow \text{num } T$
$T \rightarrow * \text{ num } T \mid \epsilon$

```
Proceduce Match(token t)
{
   if lookahead=t
   lookahead=next_token;
   else
      Error();
}

Procedure Error
{
         print("Error");
}
```

```
Procedure E
{
      if lookahead=num
      {
            Match(num);
            T();
      }
      else
            Error();
      if lookahead=$
      {
            Declare success;
      }
      else
            Error();
}
```

```
Procedure T
{
      if lookahead='*'
      {
            Match('*');
            if lookahead=num
            {
                  Match(num);
                  T();
            }
            else
                  Error();

      }
      else
            NULL
}
```

3	*	4	$

Success

3	4	*	$

Recursive Descent Parser Exercise

$X \rightarrow charY$

$Y \rightarrow +charY \mid \epsilon$

```
Procedure X
{
If lookahead=char
{
  Match (char);
  Y();
}
Else
   Error();
If lookahead= $
{
Declare Success;
}
Else
Error();
}
```

```
Procedure Y
{
If lookahead='+'
{
  Match ('+');
   If lookahead= char
{
Match (char);
Y();
}
Else
   Error();
Else
       NULL
}
```

Recursive Descent Parser

- **Advantages:**
  - They are exceptionally simple
  - They can be constructed from recognizers simply by doing some extra work specifically, building a parse tree
- **Disadvantages:**
  - Not very fast.
  - It is difficult to provide really good error messages
  - They cannot do parses that require arbitrarily long lookaheads

3. Predictive Parser

- Special case of Recursive descent parser is called Predictive parser.
- In Many cases, by carefully writing a grammar, eliminating left recursion from it, and left factoring the resulting grammar, we can obtain a grammar that can be parsed by a recursive descent parser that needs no backtracking i.e. Predictive parser.

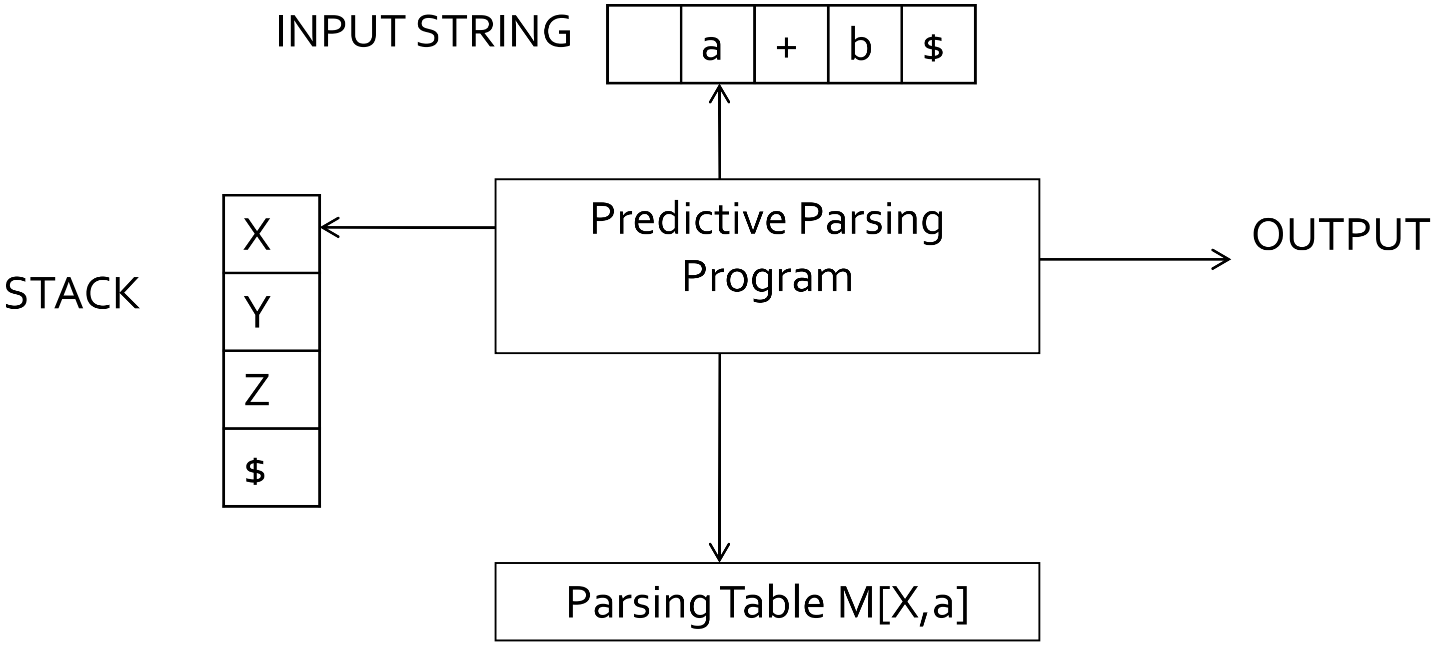
stmt → **if** expr **then** stmt **else** stmt

| **while** expr **do** stmt

| **begin** stmt_list **end**

- The keywords **if, while** and **begin** tell us which alternative is the only one that could possibly succeed if we are to find a statement.

Non Recursive Predictive parsing



Non Recursive Predictive parsing

- The parser works as follow:

1) If X = a = \$, the parser halts and announces successful completion of parsing
2) If X = a ≠ \$, the parser pops X off the stack and advance input pointer to the next input symbol.
3) If X is Non terminal, the program consults entry M[X,a] of parsing Table M. This entry will be either X-production of grammar or an error entry.

   For eg : M [X,a] = { X → U V W },

The parser replace X on top of stack by WVU (With U on top).

If M[X,a] = Error

The Parser Calls an error recovery routine.

LL(1) Parser or Predictive parsing

- An LL(1) parser is a table driven parser.
- First L stands for left-to-right scanning, Second L stands for Left most derivation and ‘1’ in LL(1) indicates that the grammar uses a look-ahead of one source symbol – that is, the prediction to be made is determined by the next source symbol.
- A major advantage of LL(1) parsing is its amenability to automatic construction by a parser generator.
- The data structure used by this parser are input buffer, stack and parsing table.

LL(1) Parser or Predictive parsing

Steps to construct LL(1)

I. Remove left recursion.(if any)
II. Remove left factoring.(if any)
III. Compute first and follow of non terminal.
IV. Construct predictive parsing table.
V. Parse the input string with the help of parsing table.

LL(1) Parser Example:

E→ E + T | T
T→ T * F | F
F→ id | (E)

- Step 1:- Remove Left Recursion

E→ TE’
E’→ +TE’ | ε
T → FT’
T’→ *FT’ | ε
F→ id | (E)

LL(1) Parser Example:

E→ E + T | T
T→ T * F | F
F→ (E) | id

Step 1:- Remove Left Recursion

E→ TE'
E'→ +TE' | ε
T → FT'
T'→ *FT' | ε
F→ (E) | id

Step 2:- Remove Left Factoring (No left factoring needed)
Step 3:- FIRST and FOLLOW

	FIRST	FOLLOW
E	{(, id}	{$,)}
E'	{+, ε}	{$,)}
T	{(, id}	{ +, $,)}
T'	{*, ε}	{+, $,)}
F	{(, id}	{*, +, $,)}

LL(1) Parser Example:

E→ E + T | T
T→ T * F | F
F→ (E) | id

Step 4: Predictive parsing table

	id	*****	**+**	**(**	**)**	**$**
E	E→TE’			E→TE’		
E’			E’→+TE’		E’→ε	E’→ε
T	T→FT’			T→FT’		
T’		T’ →*FT’	T’→ε		T’ →ε	T’→ε
F	F→id			F→(E)		

This grammer is LL(1) parser because table has no multiple defined entries.

LL(1) Parser Example:

E→ E + T | T
T→ T * F | F
F→ (E) | id

String:
id+id*id$

Step 5: Parse the input string with the help of parsing table.

Stack	Input	Action
$E	id+id*id$	
$E'T	id+id*id$	E→TE'
$E'T'F	id+id*id$	T→FT'
$E'T'id	id+id*id$	F→id
$E'T'	+id*id$	
$E'	+id*id$	T'→ ε
$E'T+	+id*id$	E'→+TE'
$E'T	id*id$	
$E'T'F	id*id$	T→FT'
$E'T'id	id*id$	F→id
$E'T'	*id$	
$E'T'F*	*id$	T'→*FT'
$E'T'F	id$	
$E'T'id	id$	F→id
$E'T'	$	
$E'	$	T'→ε
$	$	E'→ε

LL(1) Parser Example:

Check whether given grammar is LL(1) or not?

1. $S \rightarrow 1AB \mid \varepsilon$

   $A \rightarrow 1AC \mid 0C$

   $B \rightarrow 0S$

   $C \rightarrow 1$

2. $S \rightarrow A \mid a$

   $A \rightarrow a$

3. $S \rightarrow (L) \mid a$

   $L \rightarrow SL'$

   $L' \rightarrow)SL' \mid \varepsilon$

4. $S \rightarrow W$

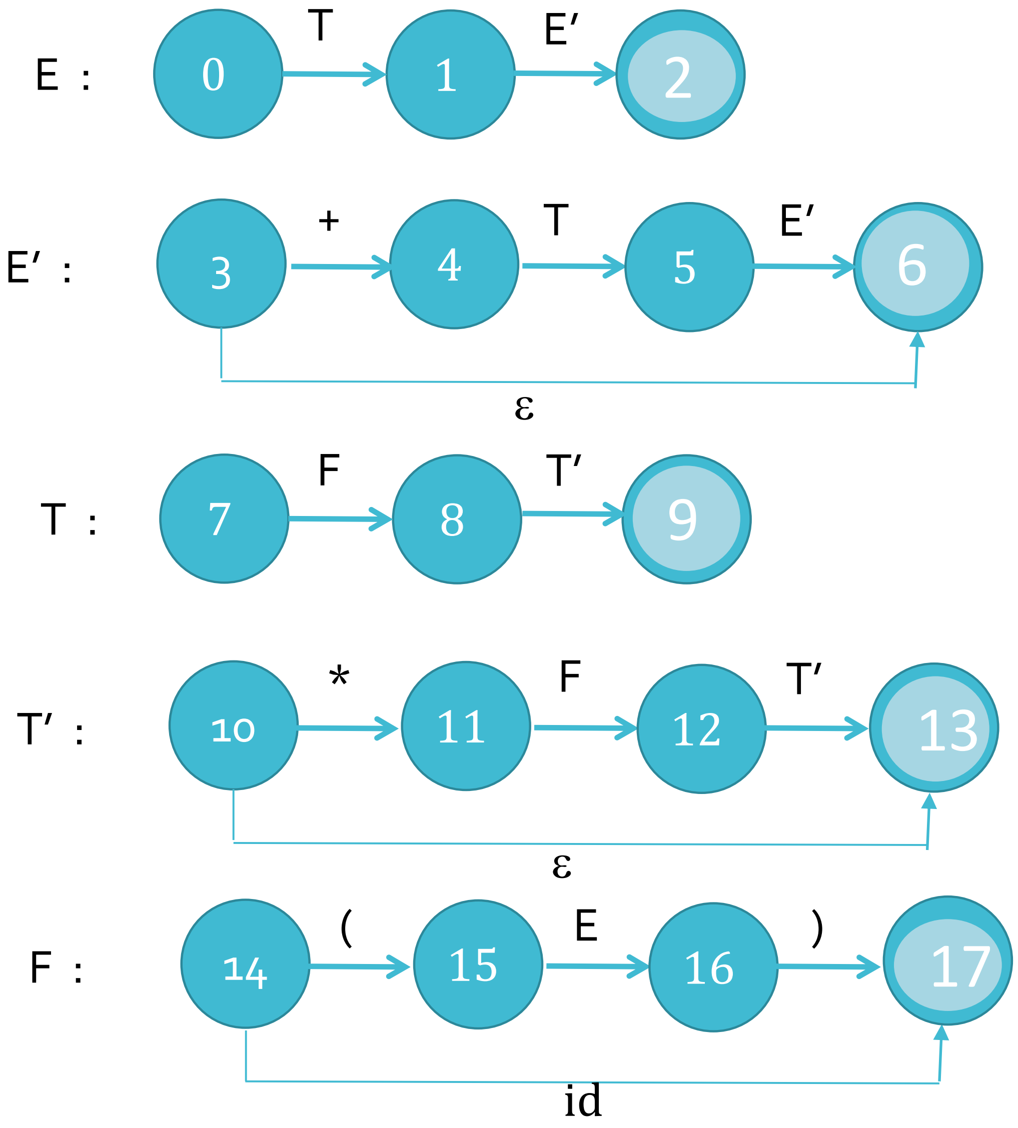
   $W \rightarrow ZXY \mid XY$

   $Y \rightarrow c \mid \varepsilon$

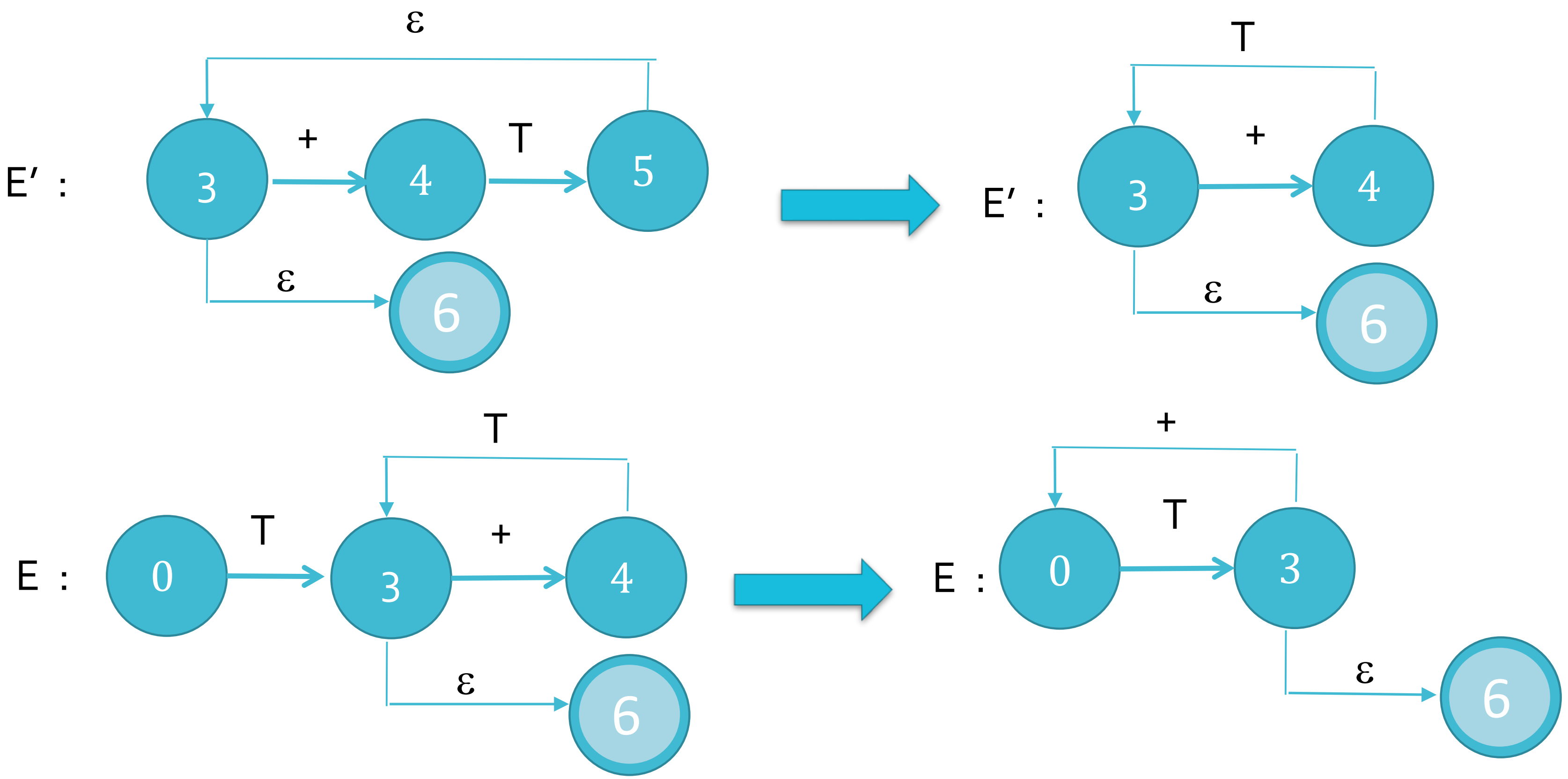
   $Z \rightarrow a \mid d$

   $X \rightarrow Xb \mid \varepsilon$

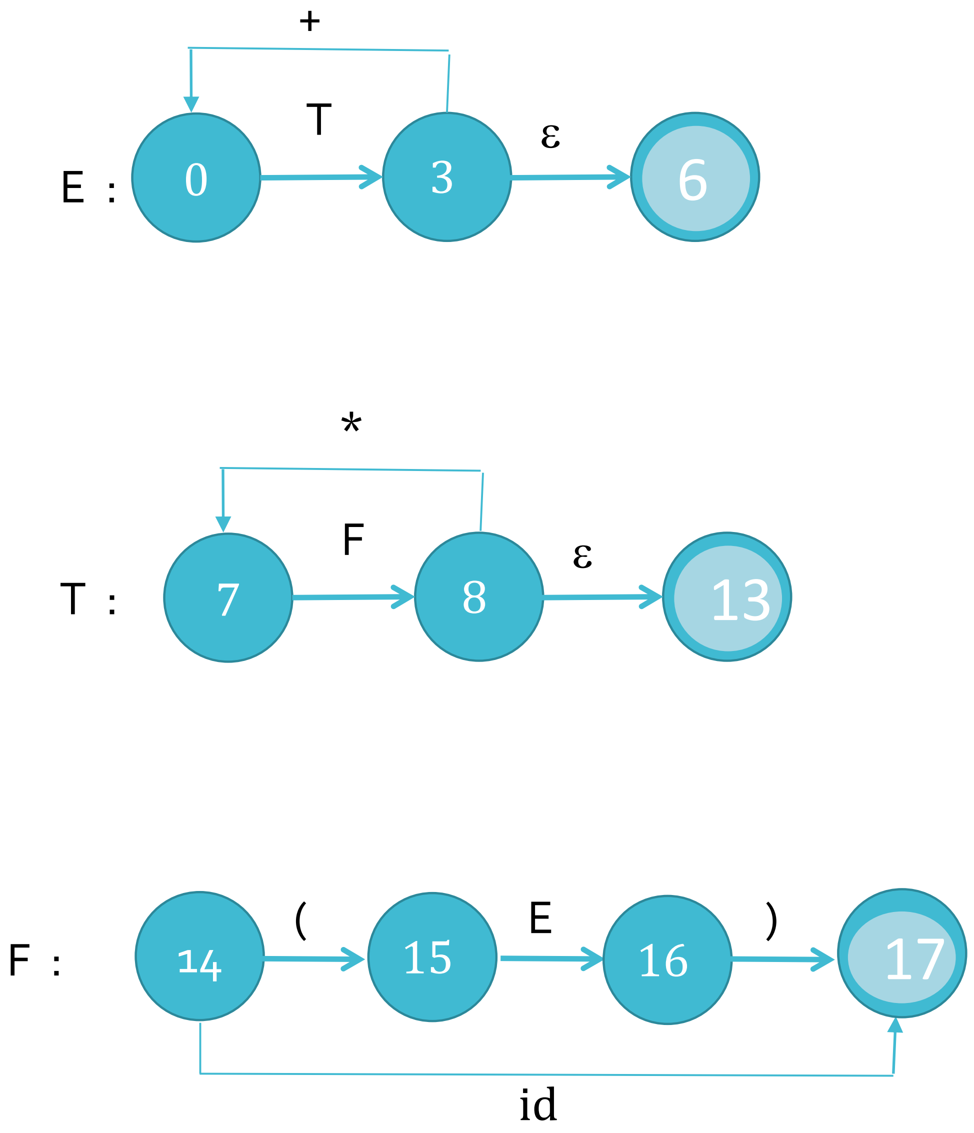
Transition Diagram for Predictive Parser



Transition Diagram for Predictive Parser



Transition Diagram for Predictive Parser



Handle and Handle Pruning

- Handle:- A handle of a string is a substring of the string that matches the right side of a production, and we can reduce such string by a non terminal on left hand side production.
- Handle Pruning:- The process of discovering a handle and reducing it to appropriate left hand side non terminal is known as handle pruning.

Right sentential form	**Handle**	**Reducing Production**
id1+id2*id3	**Id1**	**E→id**
E+id2*id3	**Id2**	**E→id**
E+E*id	**Id3**	**E→id**
E+E*E	**E*E**	**E→E*E**
E+E	**E+E**	**E→E+E**
E		

E -> E + E
E -> E * E
E -> id

id + id == >string
|
v
E + id (id is handle)

E + E (id is handle)

E (E+E is handle)

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).
- “Reducing” a string w to the start symbol of a grammar.
- At each step, decide on some substring that matches the RHS of some production.
    - Replace this string by the LHS (called **reduction**).

Shift Reduce Parsing

- It has following operations:

1. **Shift:-** Moving of the symbols from input buffer onto the stack, this action is called shift.
2. **Reduce:-** If the handle appears on the top of the stack then reduction of it by appropriate rule is done. That means R.H.S of the rule is popped of and L.H.S is pushed in. This action is called Reduce action.
3. **Accept:-** If the stack contains start symbol only and input buffer is empty at the same time then that action is called accept.
4. **Error:-** A situation in which parser cannot either shift or reduce the symbol, it cannot even perform the accept action is called as error.

Shift Reduce Parsing

Example:

- $E \rightarrow E + T \mid T$
- $T \rightarrow T * F \mid F$
- $F \rightarrow id$

String : id + id * id

Shift Reduce Parsing

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

Shift Reduce Parsing

- Example:

1. E → E - E

   E → E * E

   E→ id

String: id-id*id

2. S→ TL;

   T→ int | float

   L → L,id | id

String: int id,id;

Sol.1

Stack	Input Buffer	Parsing Action
$	id – id x id $	Shift
$ id	– id x id $	Reduce E → id
$ E	– id x id $	Shift
$ E –	id x id $	Shift
$ E – id	x id $	Reduce E → id
$ E – E	x id $	Shift
$ E – E x	id $	Shift
$ E – E x id	$	Reduce E → id
$ E – E x E	$	Reduce E → E x E
$ E – E	$	Reduce E → E – E
$ E	$	Accept

Sol.2

Stack	Input Buffer	Parsing Action
$	int id , id ; $	Shift
$ int	id , id ; $	Reduce T → int
$ T	id , id ; $	Shift
$ T id	, id ; $	Reduce L → id
$ T L	, id ; $	Shift
$ T L ,	id ; $	Shift
$ T L , id	; $	Reduce L → L , id
$ T L	; $	Shift
$ T L ;	$	Reduce S → T L
$ S	$	Accept

Shift Reduce Parsing

Example 3 – Consider the grammar

E –> 2E2

E –> 3E3

E –> 4

Perform Shift Reduce parsing for input string “32423”.

Stack	Input Buffer	Parsing Action
$	32423$	Shift
$3	2423$	Shift
$32	423$	Shift
$324	23$	Reduce by E --> 4
$32E	23$	Shift
$32E2	3$	Reduce by E --> 2E2
$3E	3$	Shift
$3E3	$	Reduce by E --> 3E3
$E	$	Accept

Operator Precedence Parsing

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

Example:-

E→AB
A→a
B→b

Not Operator Precedence

E→EOE
E→id
O→+ | -

Not Operator Precedence

E→ E + E
E→ E - E
E→id

Operator Precedence

Operator Precedence Parsing

In operator precedence parsing,

- Firstly, we define precedence relations between every pair of terminal symbols.
- Secondly, we construct an operator precedence table.

In Operator Precedence Parsing , we define following relations.

Relation	Meaning
a < b	“a” has lower precedence than terminal “b”
a = b	a “has the same precedence as” b
a > b	“a” has higher precedence than terminal “b”

Precedence Table

A precedence table is used in operator precedence parsing to establish the relative precedence of operators and to resolve shift-reduce conflicts during the parsing process. The table instructs the parser when to shift (consume the input and proceed to the next token) and when to reduce (apply a production rule to reduce a set of tokens to a non-terminal symbol).

Rules:

1.Id has highest precedence of all

2. $ has lowest precedence of all

3. If two operators have same precedence then we check the associativity.

Operator	Precedence	Associative
↑	1	right
*, /	2	left
+, -	3	left

Operator Precedence Parsing

1. Set i pointer to first symbol of string w. The string will be represented as follows

2. If $ is on the top of the stack and if a is the symbol pointed by i then return.
3. If a is on the top of the stack and if the symbol b is read by pointer i then
   a) if a <. b or a = b then

      push b on to the stack

      advance the pointer i to next input symbol

   b) Else if a .> b then

      While (top symbol of the stack .> recently popped terminal symbol)

      {

      Pop the stack. Here popping the symbol means reducing the terminal symbol by equivalent non terminal.

      }

   c) Else error()

Operator Precedence Example

Example: E→ E+E | E*E | id

String: id+id*id

Step:1 Operator Precedence Table

	id	+	x	$
id		>	>	>
+	<	>	<	>
x	<	>	>	>
$	<	<	<	

Step:2 Parse the input string

Operator Precedence Parsing

Stack	Input	Action
$	id+id*id$	$<. id Shift
$id	+id*id$	id.>+ Pop Stack
$	+id*id$	Shift
$+	id*id$	Shift
$+id	*id$	id .> * Pop Stack
$+	*id$	Shift
$+*	id$	Shift
$+*id	$	id .> $ Pop Stack
$+*	$	*.> $ Pop Stack
$+	$	+ .> $ Pop Stack
$	$	accept

Operator Precedence Parsing

Handle can be found by following process :

1) Scan the string from left end until the 1st greater than is encountered.

2) Then scan backwards (to the left) over any equals to (=) until less than is encountered.

3) The handle contains everything to the left of 1st > and to the right of < encountered in step 2. Including any surrounding non terminal.

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

E -> E+E | E*E | id

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

Precedence function

g

b

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

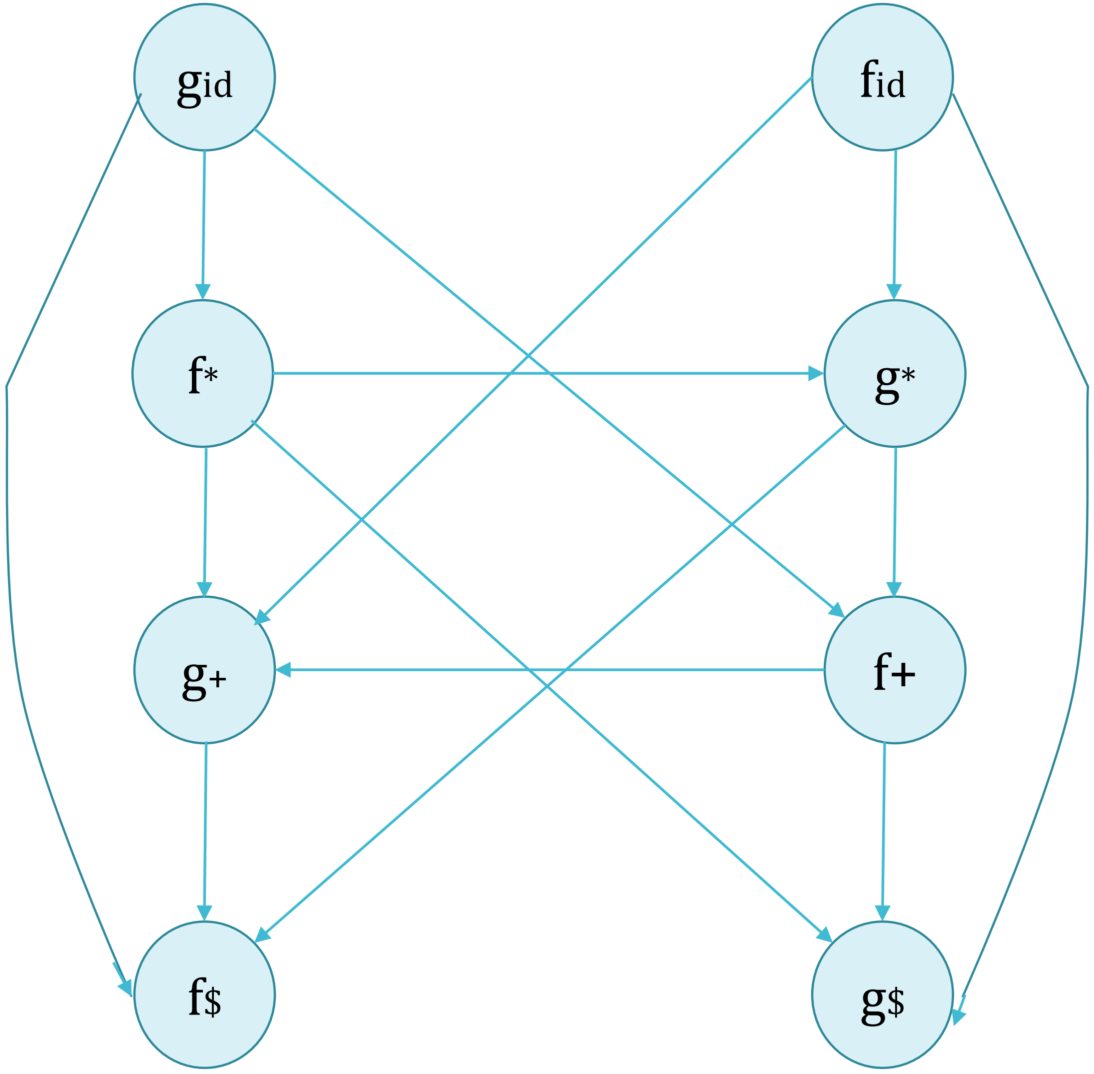
f

a

If a .> b, then fa $\longrightarrow$ gb
If a <. b, then fa $\longleftarrow$ gb

Precedence function

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



g_{id} .> f_* $\quad$ $g_{id} \rightarrow f_*$

gid .> f+ $\quad$ $g_{id} \rightarrow f+$

gid .> f+ $\quad$ $g_{id} \rightarrow f\$$

fid .> g+ $\quad$ $f_{id} \rightarrow g+$

fid .> g* $\quad$ $f_{id} \rightarrow g*$

fid .> g$ $\quad$ $f_{id} \rightarrow g\ \$$

*Same way consider for all

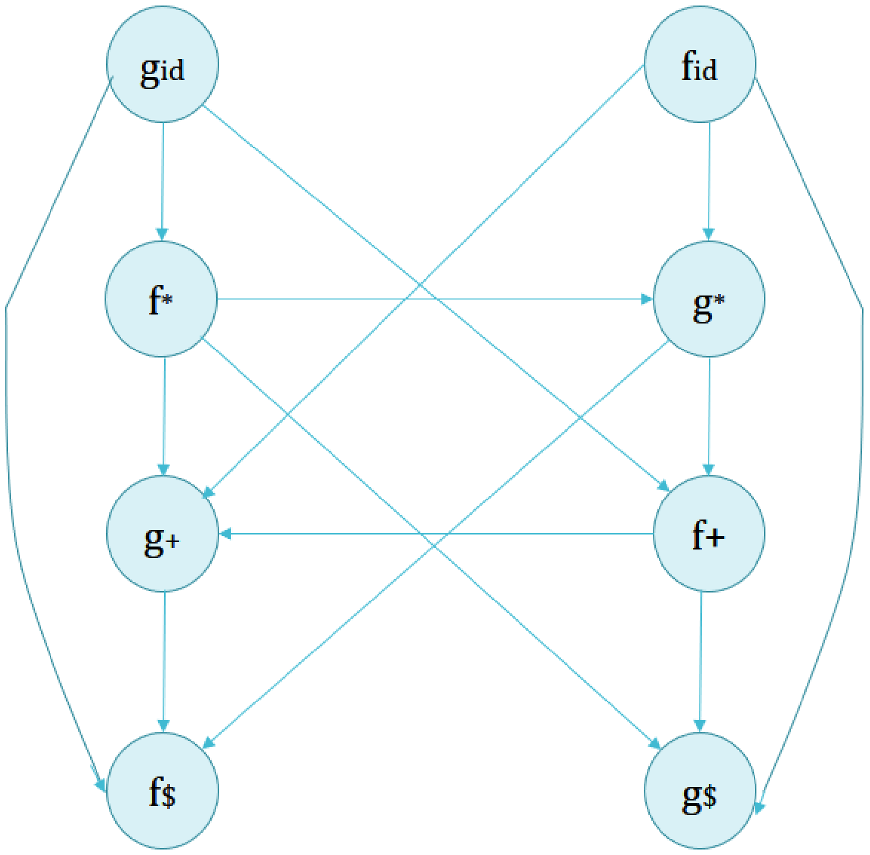
Precedence function

If the constructed graph has an cycle then no precedence function exist.
When there are no cycles collect the length of the longest paths from the group of f_a and g_b respectively.

Longest Path are:

$f_{id} \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$

$g_{id} \rightarrow f_* \rightarrow g_* \rightarrow f_* \rightarrow g_+ \rightarrow f_\$$



Now,

	+	*	id	$
f	2	4	4	0
g	1	3	5	0

Operator Precedence Parsing

	+	-	*	/	^	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
-	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
^	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=·	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Operator Precedence Parsing

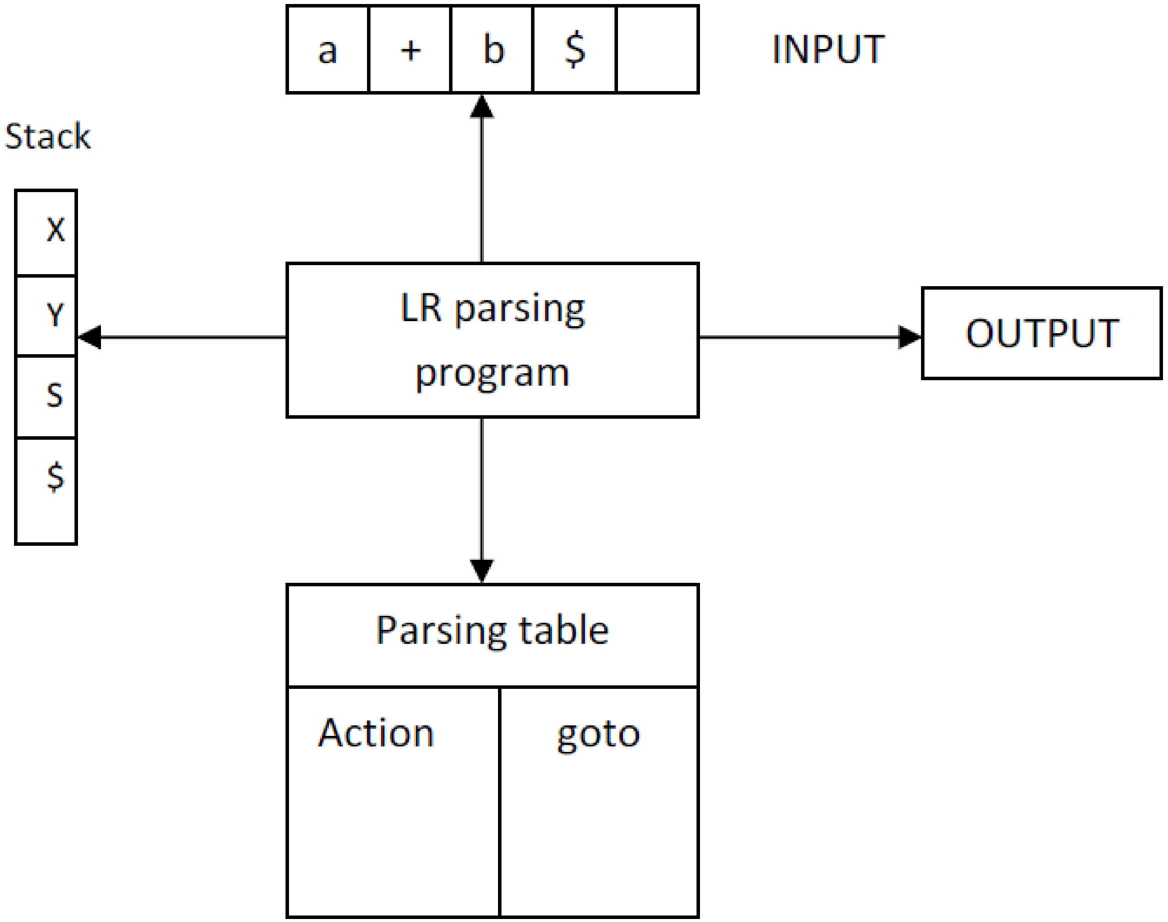
- **Disadvantages**:
  - Small class of grammars.
  - Difficult to decide which language is recognized by the grammar.

- **Advantages**:
  - simple
  - powerful enough for expressions in programming languages

LR Parsers

- This is the most efficient method of the bottom-up parsing which can be used to parse the large class of context free grammars. This method is also called LR parser.
- Non backtracking shift reduce technique.
- L stands for Left to right scanning
- R stands for rightmost derivation in reverse.

LR Parsers



a
+
b
$
INPUT
Stack
X
Y
S
$
LR parsing program
OUTPUT
Parsing table
Action
goto

LR Parsers

The structure of LR parser consists of input buffer for storing the input string, a stack for storing the grammar symbols, output and a parsing table comprised of two parts, namely actions and goto.

There is one parsing program which is actually driving program and reads the input symbol out at a time from the input buffer.

The driving program works on following line.

1. It initializes the stack with start symbol and invokes scanner (lexical analyzer) to get next token.
2. It determines s_j the state currently on the top of the stack and a_i the current input symbol.
3. It consults the parsing table for the action $\{s_j, a_i\}$ which can have one of the values.

LR Parsers

i. s_i means shift state i.
ii. r_j means reduce by rule j.
iii. Accept means Successful parsing is done.
iv. Error indicates syntactical error.

Definitions

LR(0):- The LR(0) item for grammar G is production rule in which symbol • is inserted at some position in RHS of the rule.

S→ • ABC , S→ A •BC , S→ ABC •

Augmented Grammar:- If a grammar G is having start symbol S then augmented grammar is a new grammar G' in which S' is a new start symbol such that S'→S. The purpose of this grammar is to indicate the acceptance of input.

Kernel items:- It is collection of items S'→ •S and all the items whose dots are not at the leftmost end of RHS of the rule.

Non kernel items:- The collection of all the items in which • are at the left end of RHS of the rule.

Functions closure and goto:- These are two important functions required to create collection of canonical set of items.

Viable Prefix:- It is the set of prefixs in the right sentential form of production A→α. This set can appear on the stack during shift/reduce action.

LR(0) Parser

Example:-

S→ AA

A→ aA | b

1st Step:-

S' → S

S → AA

A → aA

A → b

2nd Step:-

I0 = S' → • S

S → •AA

A → •aA

A → •b

I1 = goto (I0 , S)

S' → S•

I2 = goto (I0, A)

S → A •A

A → •aA

A → •b

I3 = goto (I0, a)

A → a • A

A → •a A

A → •b

I4 = goto (I0, b)

A → b •

I5 = goto (I2, A)

S → AA •

=goto (I2, a)

same as I3

=goto (I2, b)

same as I4

I6 = goto (I3, A)

A → aA•

= goto (I3, a)

Same as I3

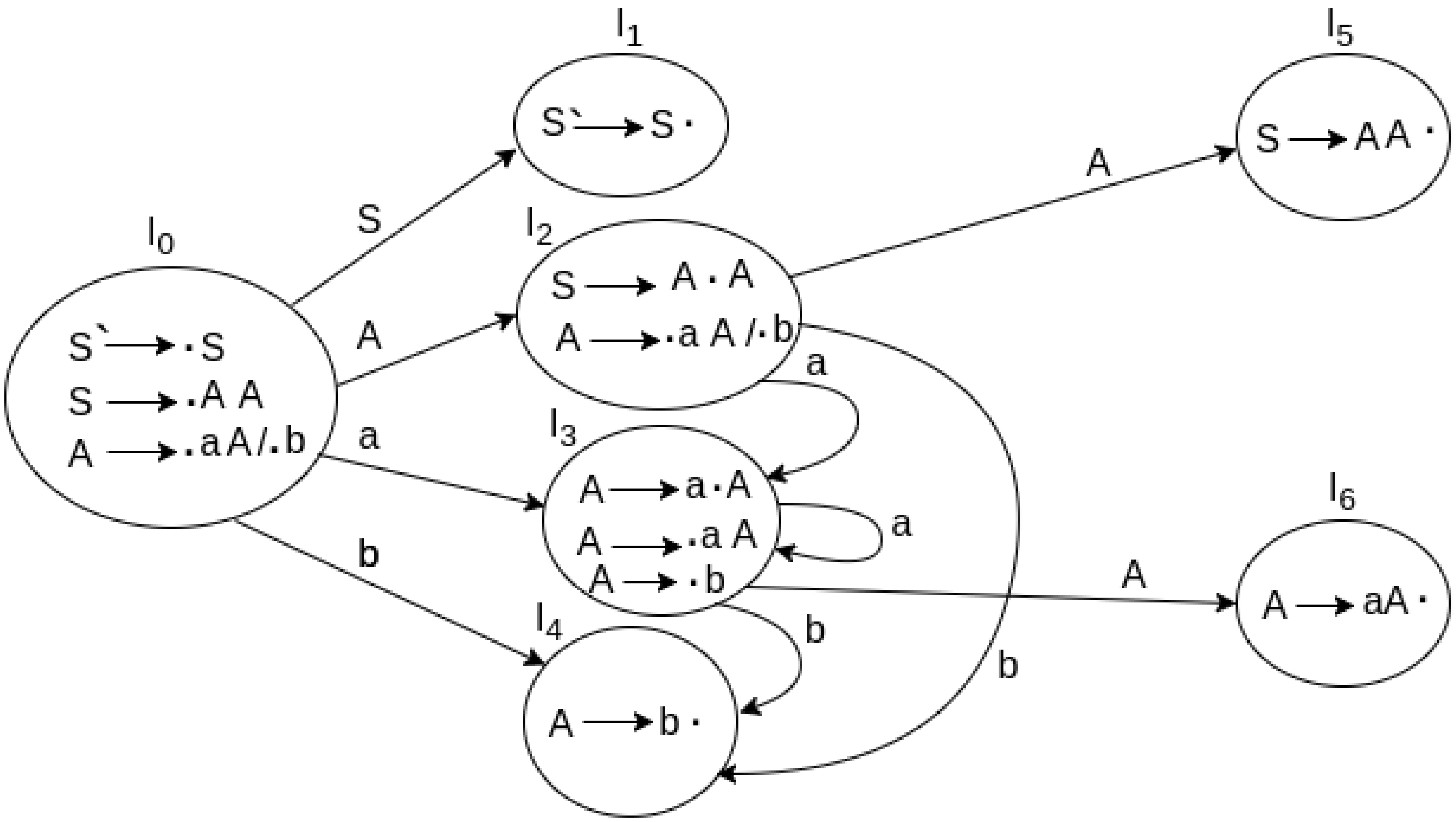
= goto (I3, b)

Same as I4

S -> oS

goto (I0,S)

LR(0) Parser



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

LR(0) Parser

Action				Goto	
	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S3	S4			5
3	S3	S4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

LR(0) Parser

Stack	Input	Action
0	aabb$	
0a3	abb$	S3
0a3a3	bb$	S3
0a3a3b4	b$	S4
0a3a3A6	b$	r3
0a3A6	b$	r2
0A2	b$	r2
0A2b4	$	S4
0A2A5	$	r3
0S1	$	r1
Accept		

LR(0) Disadvantage

Draw a parsing table for the following grammer:

S→ aSa | bSb | c

LR(0) Disadvantage

States

q_0 :	$S' \to \bullet S\$$ $S \to \bullet aSa$ $S \to \bullet bSb$ $S \to \bullet c$
q_1 :	$S' \to S \bullet \$$
q_2 :	$S \to a \bullet Sa$ $S \to \bullet aSa$ $S \to \bullet bSb$ $S \to \bullet c$
q_3 :	$S \to b \bullet Sb$ $S \to \bullet aSa$ $S \to \bullet bSb$ $S \to \bullet c$
q_4 :	$S \to c\bullet$

LR(0) Disadvantage

States

q_5 :	$S \to aS \bullet a$
q_6 :	$S \to bS \bullet b$
q_7 :	$S \to aSa\bullet$
q_8 :	$S \to bSb\bullet$

LR(0) Disadvantage

State	Action				Goto
	a	b	c	$\$$	S
0	s_2	s_3	s_4		1
1				*accept*	
2	s_2	s_3	s_4		5
3	s_2	s_3	s_4		6
4	r_3	r_3	r_3	r_3	
5	s_7				
6		s_8			
7	r_1	r_1	r_1	r_1	
8	r_2	r_2	r_2	r_2	

LR(0) Disadvantage

- In LR(0) the reduction string is in entire row. Therefore, we have to reduce by taking the decision looking at grammar.
- So, it is not powerful and accurate.

SLR

- SLR means simple LR.
- A grammar for which an SLR parser can be constructed is said to be an SLR.
- SLR is a type of LR parser with small parse tables and a relatively simple parser generator algorithm.
- The parsing table has two states(action, goto)
- The parsing table has values:

1. Shift S, where S is a state
2. Reduce by a grammar production
3. Accept and
4. Error

SLR(1) Parser

Example:-

S→ CC

C→ cC | d

1st Step:-

S' → S

S → CC

C→ cC

C→ d

2nd Step:-

I0 = S' → • S

S → •CC

C → •cC

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 → •c C

C → •d

I4 = goto (I0, d)

C → d•

I5 = goto (I2, C)

S → CC •

=goto (I2, c)

same as I3

=goto (I2, d)

same as I4

I6 = goto (I3, C)

C → cC•

= goto (I3, c)

Same as I3

= goto (I3, d)

Same as I4

SLR(1) Parser

Action				Goto	
	c	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S3	S4			5
3	S3	S4			6
4	r3	r3	r3		
5			r1		
6	r2	r2	r2		

Follow:

S' = {$}

S = {$}

C = {c,d,$}

0 S' → S

1 S → CC

2 C→ cC

3 C→ d

SLR(1) Parser

Stack	Input	Action
0	dcd$	
0d4	cd$	S3
0C2	cd$	R3
0C2c3	d$	S3
0C2c3d4	$	S4
0C2c3C6	$	R3
0C2C5	$	R2
0S1	$	R1
Accept		

SLR(1) Parser

Check whether given grammar is SLR or not.

Example 1:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

String : id+id*id

Example 2:

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

Example 3 :

$S \rightarrow L=R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

SLR(1) Parser

Example:-

E → E + T | T

T → T * F | F

F → (E) | id

1st Step:-

E' → E

E → E + T | T

T → T * F | F

F → (E) | id

2nd Step:-

I0 = E' → •E

E → •E+T | •T

T → •T*F | •F

F →•(E) | •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 | •T

T → •T*F | •F

F →•(E) | •id

I5 = goto (I0, id)

F → id•

I6=goto (I1, +)

E →E+•T

T → •T*F | •F

F →•(E) | •id

I7 = goto (I2, *)

T →T*•F

F →•(E) | •id

I8 = goto (I4, E)

F → (E•)

E → E•+T

= goto (I4, T)

Same as I2

= goto (I4, F)

Same as I3

= goto (I4, ( )

Same as I4

SLR(1) Parser

I4 = goto (I0, ( )

F → (•E)

E → •E+T | •T

T → •T*F | •F

F →•(E) | •id

I6=goto (I1, +)

E →E+•T

T → •T*F | •F

F →•(E) | •id

I7 = goto (I2, *)

T →T*•F

F →•(E) | •id

I8 = goto (I4, E)

F → (E•)

E → E•+T

= goto (I4, id)

Same as I5

I9 = goto (I6, T)

E → E+T•

T → T•*F

= goto (I6, F)

Same as I3

= goto (I6, ( )

Same as I4

= goto (I6, id)

Same as I5

I10 = goto (I7, F)

T → T*F•

= goto (I7, ( )

Same as I4

= goto (I7, id)

Same as I5

I11 = goto (I8,))

F →(E)•

= goto (I8, +)

Same as I6

= goto (I9, *)

Same as I7

FOLLOW (E') = {$}

FOLLOW (E) = { +,),$ }

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

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

SLR(1) Parser

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			

SLR(1) Parser

Example :-

S → L = R | R

L → * R | id

R → L

1st Step:-

S' → S

S → L = R | R

L → * R | id

R → L

2nd Step:-

I0 = S' → •S

S → • L = R | • R

L → • * R | • 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

R → • L

L → • * R | • id

I5 = goto (I0, id)

L → id•

I6=goto (I2, =)

S → L = • R

R → • L

L → • * R | • id

I7 = goto (I4, R)

L → * R•

= goto (I4, *)

Same as I4

I8 = goto (I4, L)

R → L •

= goto (I4, id)

Same as I5

I9= goto (I6, R)

S → L = R•

= goto (I6, L)

Same as I8

= goto (I6, *)

Same as I4

= goto (I6, id)

Same as I5

SLR(1) Parser

Example :-

S → AaAb

S → BbBa

A → ε

B → ε

1st Step:-

S' → S

S → AaAb

S → BbBa

A → ε

B → ε

2nd Step:-

I0 = S' → •S

S → • AaAb | • BbBa

A → •

B → •

I1 = goto (I0, S)

S' → S•

I2 = goto (I0, A)

S → A • a A b

I3 = goto (I0, B)

S → B • b B a

I4 = goto (I2, a)

S → A a • A b

A → •

I5 = goto (I3, b)

S → B b • B a

B → •

I6= goto (I4, A)

S → A a A • b

I7 = goto (I5, B)

S → B b B • a

I8 = goto (I6, b)

S → A a A b •

I9 = goto (I7, a)

S → B b B a •

FOLLOW (S') = {$}

FOLLOW (S) = {$}

FOLLOW (A) = {a,b}

FOLLOW (B) = {a,b}

SLR(1) Parser

ACTION				GOTO		
	a	b	$	S	A	B
0	R3/R4	R3/R4		1	2	3
1			Accept			
2	S4					
3		S5				
4	R3	R3			6	
5	R4	R4				7
6		S8				
7	S9					
8			R1			
9			R2			

CLR(1)

- Canonical LR(1) parser.
- CLR(1) Parsing configurations have the general form:
- The Look Ahead Component 'a' represents a possible look-ahead after the entire right-hand side has been matched.
- CLR parser is the **most powerful parser**.

LR(1) item = LR(0) item + lookahead

LR(0) Item .... $A \rightarrow \alpha \bullet \beta$

LR(1) Item .... $A \rightarrow \alpha \bullet \beta , a$

In LR(1) "a" is lookahead symbol

$A \rightarrow \alpha \bullet B\beta , a \quad \Rightarrow \quad \text{FIRST}(\beta , a)$

CLR(1) Parser

Example:-

S→ CC

C→ aC | d

1st Step:-

S' → S

S → CC

C→ aC

C→ d

2nd Step:-

I0 = S' → • S, $

S → •CC, $

C → •aC , a/d

C →•d, a/d

I1 = goto (I0 , S)

S' → S•,$

I2 = goto (I0, C)

S → C •C, $

C → •aC, $

C → •d, $

I3 = goto (I0, a)

C → a • C, a/d

C → •a C, a/d

C → •d, a/d

I4 = goto (I0, d)

C → d•, a/d

I5 = goto (I2, C)

S → CC •, $

I6 = goto (I2,a)

C → a•C, $

C → •aC,$

C → •d,$

I7=goto(I2, d)

C →d•, $

I8 = goto (I3, C)

C → aC•, a/d

goto (I3, a)

= same as I3

goto (I3, d)

= same as I4

I9 = goto (I6, C)

C → aC•, $

goto (I6, a)

= Same as I6

goto (I6, d)

= Same as I7

CLR(1) Parser

Action				Goto	
	a	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

Reduce entry only in the lookahead.

CLR(1) Parser

Stack	Input	Action
0	aadd$	
0a3	add$	S3
0a3a3	dd$	S3
0a3a3d4	d$	S4
0a3a3C8	d$	R3
0a3C8	d$	R2
0C2	d$	R2
0C2d7	$	S7
0C2C5	$	R3
0S1	$	R1
Accept		

CLR(1)

- Example:-

$S \rightarrow L = R \mid R$

$L \rightarrow * R \mid id$

$R \rightarrow L$

Parse String : *id=id

LALR(1)

- In this type of grammar the lookahead symbol is generated for each set of item. The table obtained by this method are smaller compared to clr(1) parser.
- In fact the state of LALR(Look Ahead LR) and SLR are always same.
- We follow the same steps as discussed in SLR and canonical LR parsing techniques and those are:

1. Construction of canonical set of items along with look ahead.
2. Building LALR parsing table.
3. Parsing the input string using canonical LR parsing table.

NOTE :

- No. of Item State in SLR = No. of Item State in LALR < No. of state in CLR
- CLR is most powerful among all LR Parsers.

LALR(1) Parser

Example:-

S→ CC

C→ aC | d

1st Step:-

S' → S

S → CC

C→ aC

C→ d

2nd Step:-

I0 = S' → • S, $

S → •CC, $

C → •aC , a/d

C →•d, a/d

I1 = goto (I0 , S)

S' → S•,$

I2 = goto (I0, C)

S → C •C, $

C → •aC, $

C → •d, $

I3 = goto (I0, a)

C → a • C, a/d

C → •a C, a/d

C → •d, a/d

I4 = goto (I0, d)

C → d•, a/d

I5 = goto (I2, C)

S → CC •, $

I6 = goto (I2,a)

C → a•C, $

C → •aC,$

C → •d,$

I7=goto(I2, d)

C →d•, $

I8 = goto (I3, C)

C → aC•, a/d

goto (I3, a)

= same as I3

goto (I3, d)

= same as I4

I9 = goto (I6, C)

C → aC•, $

goto (I6, a)

= Same as I6

goto (I6, d)

= Same as I7

CLR(1) Parser

	Action			Goto	
	a	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

Reduce entry only in the lookahead.

LALR(1) Parser

Action				Goto	
	a	d	$	S	C
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Reduce entry only in the lookahead.

LALR(1) Parser

Stack	Input	Action
0	aadd$	
0a36	add$	S36
0a36a36	dd$	S36
0a36a36d47	d$	S47
0a36a36C89	d$	R3
0a36C89	d$	R2
0C2	d$	R2
0C2d47	$	S47
0C2C5	$	R3
0S1	$	R1
Accept		

LALR(1) Parser

Example:-

S→Aa

S→bAc

S→dc

S→bda

A→d

In above example no state are having common production rule. Hence no states can be merged, so generate LALR(1) Parsing table for all given states.

Practise Questions: Left Factoring

1. Do left factoring in the following grammar-

$S \rightarrow aSSbS \,/\, aSaSb \,/\, abb \,/\, b$

2. Do left factoring in the following grammar-

$S \rightarrow a \,/\, ab \,/\, abc \,/\, abcd$

3. Do left factoring in the following grammar-

$S \rightarrow aAd \,/\, aB$

$A \rightarrow a \,/\, ab$

$B \rightarrow ccd \,/\, ddc$

Left Recursion practise

1. Consider the following grammar and eliminate left recursion

$A \longrightarrow ABd \ / \ Aa \ / \ a$

$B \longrightarrow Be \ / \ b$

2. Consider the following grammar and eliminate left recursion

$E \longrightarrow E + E \ / \ E \times E \ / \ a$

3. Consider the following grammar and eliminate left recursion

$S \longrightarrow (L) \ / \ a$

$L \longrightarrow L \ , S \ / \ S$

Practise Questions: Ambiguity

1.Prove wheter the grammer is ambiguous or not:

Consider the following grammar-

$S \longrightarrow ABC$

$A \longrightarrow a$

$B \longrightarrow b$

$C \longrightarrow c$

Consider a string w = abc.

2. Use G be the grammar

$S \rightarrow aB \mid bA$

$A \rightarrow a \mid aS \mid bAA$ $B \rightarrow b \mid bS \mid aBB$

For the string aaabbabbba ,

Find

a. Leftmost Derivation.

b. Rightmost Derivation.

c. Derivation Tree

Practise Questions: first,follow

1.Calculate the first and follow functions for the given grammar-

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC \mid \in$

$D \rightarrow EF$

$E \rightarrow g \mid \in$

$F \rightarrow f \mid \in$

2.Calculate the first and follow functions for the given grammar-

$S \rightarrow A$

$A \rightarrow aB \mid Ad$

$B \rightarrow b$

$C \rightarrow g$

Thanks

Marwadi University

Prof. Shilpa Singhal