Short Questions:

1. Solve Left factorial with an example

In general if $A \to \alpha\beta_1 \mid \alpha\beta_2$ is a production then it is not possible for us to take a decision whether to choose first rule or second.

In such a situation the above grammar can be left factored as

$$A \to \alpha A'$$
$$A' \to \beta_1 \mid \beta_2$$

For eg:- $S \to iE+S \mid iE+SeS \mid a$

(α = $iE+S$, β = eS)

$$E \to b$$

The left factored grammar becomes

$$S \to iE+SS' \mid a$$
$$S' \to eS \mid \varepsilon$$
$$E \to b$$

---

2. Create Recursive Descent parsing with an example.

A parser that uses collection of Recursive Procedures for parsing the given I/p string is called Recursive Descent (RD) Parser.

- The parser should be activated by a procedure corresponding to the start symbol.

E → num T

T → * num T | ε

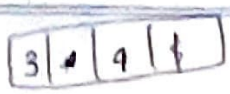
```
Procedure E
{
    if lookahead = num then
      {
          match (num);
            T;
      }
    else
        error;
    if lookahead = '$'
      {
          declare success;
      }
    else
        error;

Procedure T
{
    if lookahead = '*'
      {
          match ('*');
```

```
if look-ahead = 'num'
    {
        match (num);
          T;
    }
else
        error;
}
else
        NULL;
}
```

```
Procedure   match (token t)
{
    if  look-ahead = t
        look-ahead = next_token;
    else
        error
}
```

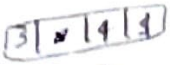
```
Procedure  error
  {
      Print (" error!! ");
  }
```



↑ T → num T

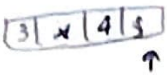
↑

T → * num T



↑

T → * num T



↑

declare success

3. Explain about SR and RR conflicts in parsing

Shift - Reduce (SR) Conflict:

- A situation in bottom-up parsing where the parser can either shift the next input symbol onto the stack or reduce the current stack of symbols by applying a production rule of the grammar.
- Typically caused by ambiguity in the grammar where a sequence of symbols could either continue as part of a larger construct or be reduced to a higher-level construct.

Reduce - Reduce (RR) Conflict:

- A situation in bottom-up parsing where the parser faces ambiguity when multiple production rules could be reduced for a given set of symbols.

- Arises from ambiguous grammars where multiple rules can apply to the same input sequence.

Long Questions :

4. Perform shift reduce parsing with an example.

- Shift reduce parser attempts to construct parse tree from leaves to root.
- Thus it works on the same principle of bottom-up parser.
- A shift reduce parser requires following Data Structures :
  1. The i/p buffer storing the i/p string
  2. A stack for storing & accessing the LHS & RHS of rules.
- The parser performs following basic operations.
  1. SHIFT : moving of the symbols from i/p buffer onto the stack.
  2. REDUCE : If the handle appears on the top of the stack then reduction of it by appropriate rule is done. That means RHS of rule is popped off and LHS is pushed in.

③ ACCEPT If the stack contains START symbol only and i/p 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 symbols, it cannot even perform the accept action is called as ERROR.

Example

$E \rightarrow E - E$

$E \rightarrow E * E$

$E \rightarrow id$

perform shift-Reduce parsing of i/p string.

id - id * id

Stack	i/p buffer	parsing Action
$	id - id * id	shift id
$ id	- id * id	Reduce $E \rightarrow id$
$ E	- id * id	shift -
$ E -	id * id	shift id
$ E - id	* id	Reduce $E \rightarrow id$
$ E - E	* id	shift *
$ E - E *	id	shift id
$ E - E * id	$	Reduce $E \rightarrow id$

$ E - E * E	$	Reduce $E \to E * E$
$ E - E	$	Reduce $E \to E - E$
$ E	$	Accept

5. Construct CLR parse table for the following grammar.

$$S \to CC$$
$$C \to aC \mid d$$

CLR passes also use an augmented grammar to facilitate passing.

The augmented grammar :

$$S' \to S$$
$$S \to CC$$
$$C \to aC \mid d$$

We Initially add $S' \to \cdot S, \$$ as the first rule in I_0.

Now match $S' \to \cdot S, \$$

$A \to \alpha \cdot X \beta, a$

$A = S', \alpha = \varepsilon, X = S, \beta = \varepsilon, a = \$$

$X \to \gamma, b \in FIRST(\beta, a)$

$S \to \cdot CC, b \in FIRST(\varepsilon \$)$

$b = \$$

$\therefore S \to \cdot CC, \$$ will be added I_0.

Now $S \to \cdot CC, \$$ is in I_0 it match

$A \to \alpha \cdot X \beta, a$

$A = S, \quad \alpha = \varepsilon, \quad X = C, \quad \beta = C, \quad a = \$$

$X \to \gamma, \quad b \in FIRST(\beta, a)$

$C \to \cdot aC, \quad b \in FIRST(C, a)$

$(a \mid d \mid \$)$

$C \to \cdot d$

$\therefore C \to \cdot aC, a \mid d$

$C \to \cdot d, a \mid d$ } added in I_0

Hence I_0 :

$S' \to \cdot S, \$$

$S \to \cdot CC, \$$

$C \to \cdot aC, a \mid d$

$C \to \cdot d, a \mid d$

→ Applying goto on I_0

goto (I_0, S)

I_1 :

$S' \to S., \$$

→ goto (I_0, C)

I_0 :

$S \rightarrow C . C , \$$

$A \rightarrow \alpha . X \beta , a$

$A = S , \alpha = C , X = C , \beta = \varepsilon , a = \$$

$X \rightarrow .\gamma , \quad b \in FIRST(\beta . a)$

$C \rightarrow .aC , \quad b \in FIRST(\epsilon . \$)$

$= \$$

$C \rightarrow .d$

$\rightarrow$ goto (I_0 , C)

I_1 : $S \rightarrow C \, C , \$$

$C \rightarrow .aC , \$$

$C \rightarrow .d , \$$

$\rightarrow$ goto (I_0 , a)

I_3 : $C \rightarrow a . C , a|d$

$C \rightarrow .aC , a|d$

$C \rightarrow .d , a|d .$

$\rightarrow$ goto (I_0 , d)

I_4 : $C \rightarrow d , a|d$

→ Applying goto on I_1,

the production is a kernel item. So we can't apply goto on I_1.

→ Applying goto on I_2

goto (I_2, C)

I_5 : $S \rightarrow CC.$, \$

goto (I_2, a)

I_6 : $C \rightarrow a.C$, \$

$C \rightarrow .aC$, \$

$C \rightarrow .d$, \$

goto (I_2, d)

I_7 : $C \rightarrow d.$, \$

→ Applying goto on I_3

goto (I_3, C)

I_8 : $C \rightarrow aC.$, a|d

goto (I_3, a)

$C \rightarrow a.C$, a|d } repetition of states I_3

goto (I_3, d)

$C \rightarrow d.$, a|d — I_4 }

→ Applying goto on I_4

Production is kernel item so goto can't be applicable.

→ Applying goto on I_5

. Its a kernel item so can't apply goto.

→ Applying goto on I_6

goto (I_6, C)

I_9 : $C \rightarrow aC.$, $

goto (I_6, a)

$C \rightarrow a.C$, $ } ... I_6 Repetition of states

goto (I_6, d)

$C \rightarrow d.$, $ ... I_7

→ I_7 kernel item

→ I_8 kernel item & repetition states

→ I_9 kernel item & repetition states.

so can't apply goto

states are from I_0 - I_9

Step 2 — Parsing table

states	ACTION			GOTO	
	a	d	$	S	C
0	S_3	S_4		1	2
1			ACCEPT		
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		

Step 3 — Parsing i/p string.

i/p "aadd"

Referring the parsing table to parse the i/p string.

stack	i/p buffer	parsing Action
$ 0	a a d d $	shift a
$ 0 a 3	a d d $	shift a
$ 0 a 3 a 3	d d $	shift d
$ 0 a 3 a 3 d 4	d $	reduce C → d
$ 0 a 3 a 3 C 8	d $	reduce C → aC
$ 0 a 3 C 8	d $	reduce C → aC
$ 0 C 2	d $	shift d
$ 0 C 2 d 7	$	reduce C → d
$ 0 C 2 C 5	$	reduce S → CC
$ 0 S 1	$	Accept