Recursive descent Parsing

lex() → called before initial parsing to get first token

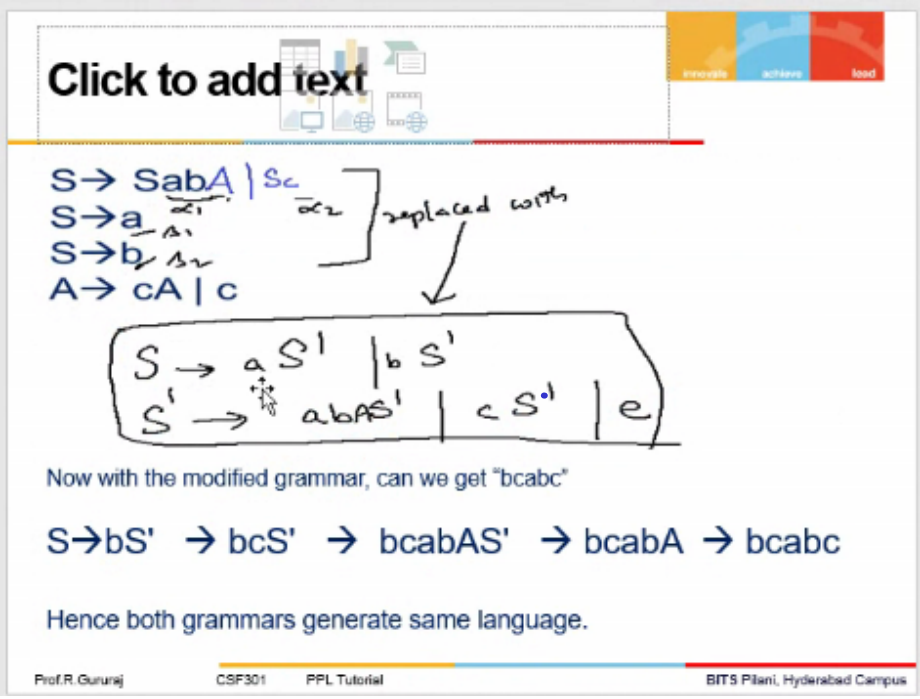
```
S()
{
  if(next token == 'a')
  {lex() A() lex()}
elsif (next token == 'b')
(accepted)

} Else
  Error
```

```
A()
{ if nextoken is b
    lex()
    A()
  elseif nexttoken is a
    lex()
    B()
```

Left recursion Eliminate

$E \rightarrow E + T$

$E \rightarrow T$



Now eliminate Left recursion from the following Grammar:

S→ SabA | Sc

S→a

S→b

A→ cA | c

Can generate: S→SabA→ScabA→bcabA→ bcabc

Whenever we have production of the form-

$A \rightarrow A\alpha_1, A \rightarrow A\alpha_2, \ldots, A \rightarrow A\alpha_n$, and

$A \rightarrow \beta_1, A \rightarrow \beta_2, \ldots, A \rightarrow \beta_m$

Replace these by

$A \rightarrow \beta_1 A', A \rightarrow \beta_2 A', \ldots, A \rightarrow \beta_m A'$, and

$A' \rightarrow \alpha_1 A', A' \rightarrow \alpha_2 A', \ldots, A' \rightarrow \alpha_n A'$, and

$A' \rightarrow e$

S → SS+

S → SS*

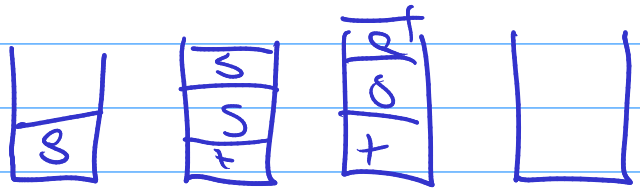
S → a

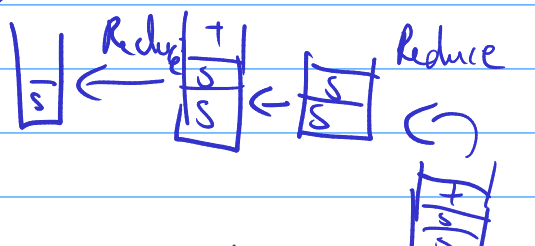
aaa*a++

→ ambiguity with SS* so we assume correct decision is taken

S→SS+ → SaS+ → aSS++

aaa*a++ ← aSS*S++



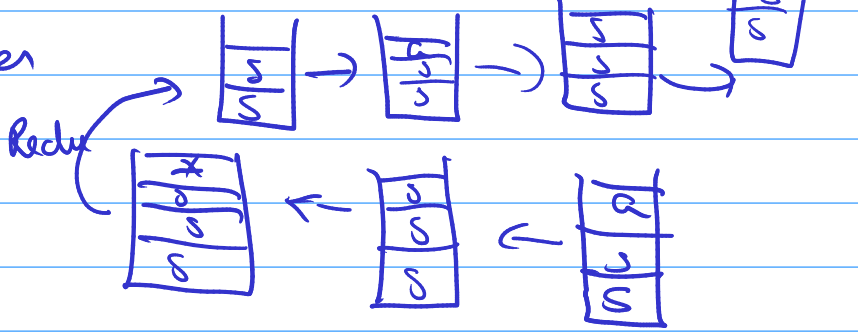


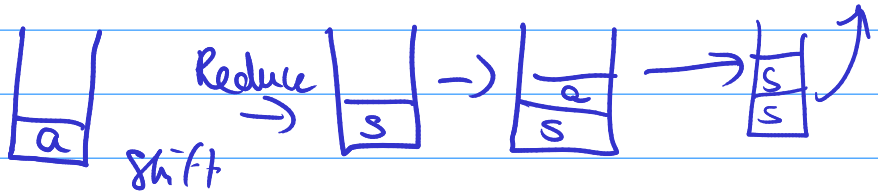
Bottom up parser

S→SS+

S→SS*

S→a





Shift or reduce?

aaa*a++

A → bb | cb , to find first token of next part and compare (first-of)

A → abb | cbb ⇒ How to resolve ambiguity ?

Ambiguities arise normally when the first symbol in RHS is same,

So we can factor it out (this is called Left factoring)

A → abb | acb

A → aM

M → bb | cb → now we can use first-of method

May or may not work

No method WILL 100% work

Topdown parsers will suffer from left recursion

Bottom up, no. :)

E ⇒ E + T