Computer Science & Engineering Department
I.I.T Kharagpur

Compilers

3rd year CSE: 5th semester (Class Test 1 Solution)

Solutions:

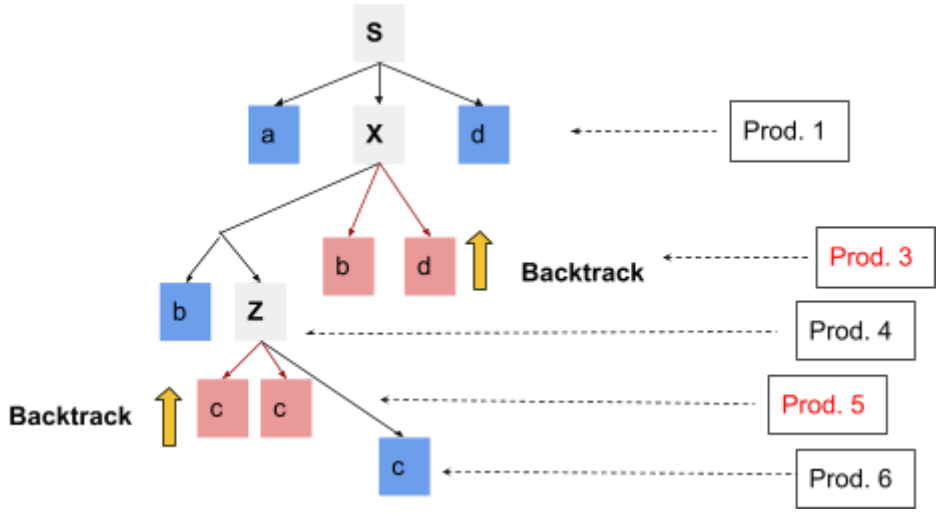
1. Parse the string **abcd** executing the non deterministic recursive descent parser with the grammar G specified below. Apply the productions strictly following the (increasing) sequence, indicated by the rule #. Clearly show the functions invoked by the parser and all backtracking steps, if any. Finally, justify if the parser accepts or rejects this string.

Rule#	Production
1.	S □ aXd
2.	S □ aZd
3.	X □ bd
4.	X □ bZ
5.	Z □ cc
6.	Z □ c

[6]

Solution:

String: abcd



- S() will go with first production ‘aXd’ ……i=1 and pointer is at ‘a’ (in abcd)
- ‘a’ is terminal and matched with the pointer value….hence i++ so i=2, pointer is at ‘b’ (in abcd)
- X() is non-terminal, invoke the function, choosing ‘bd’ production
- here ‘b’ is terminal and matched……i=3, pointer is at ‘c’ (in abcd)
- ‘c’ and ‘d’ did not match so **backtrack** ……i=2, pointer is at ‘b’ (in abcd)
- X() will choose another production ‘bZ’
- ‘b’ matches with ‘b’ in string……i=3, pointer is at ‘c’ (in abcd)
- Z() is non-terminal, invoke the function, choosing ‘cc’
- ‘c’ matches with ‘c’ in string ……..i=4, pointer is at ‘d’ (in abcd)
- another ‘c’ does not match with d in string so **backtrack….**..i=3, pointer is at ‘c’ (in abcd)
- Z() will choose ‘c’ now
- ‘c’ matches with ‘c’ in string……i=4, pointer is at ‘d’ (in abcd)
- Z() returns to X(), it returns to S(), last d matches with ‘d’ in string ... .Parsed!

2. Consider the following grammar G:

S→W

W→ZXY / XY

Y→c/ϵ

Z→a/d

X→Xb/ϵ

Justify with argument, whether the grammar G is an LL(1) grammar or not, without constructing the parsing table.

Solution:

Answer: this is not LL(1) grammar.

First(S) = { a, d, b, c, d, ϵ} , Follow(S) = { $ }

First(W) = { a, d, b, c, d, ϵ} , Follow(W) = { $ }

First(X) = { b, ϵ} , Follow(X) = {b, c, $ }

First(Y) = { c, ϵ} , Follow(Y) = { $ }

First(Z) = { a, d} , Follow(Z) = { b, c, $ }

Logic: first(X) and follow(X) are not disjoint sets violating the property-3.

{ b, ϵ} ∩ {b, c, $ } = ∅

Also, X→Xb/ϵ is left recursive hence not LL(1).

3. Consider the following grammar with two missing productions:

S □ aS | Prod 1
A □ Prod 2 | ϵ
X □ cS | ϵ
Y □ dS | ϵ
Z □ eS

Terminal={a,b,c,d,e}, NonTerminal={S,A,X,Y,Z}. S is the start symbol.
Fortunately, we have the First and Follow sets for this grammar:

	First	**Follow**
S	*{a, b, c, d, e}*	*{$}* U *Follow(X)* U *Follow(Y)* U *Follow(Z)*
A	*{c, d, e, ϵ}*	*{b}*
X	*{c, ϵ}*	*{First(Y)/ϵ}* U *First(Z)*
Y	*{d, ϵ}*	*First(Z)*
Z	*{e}*	*Follow(A)*

Reconstruct the grammar by filling in the missing two productions (Prod 1) and (Prod 2).

Solution:
Prod 1: Ab
Prod 2: XYZ
Logic:
From the First and Follow sets:
First(S) includes {a, b, c, d, e}, which indicates that S can start with 'a', 'b', 'c', 'd', 'e'.
Follow(S) includes {b, c, d, e, $}, suggesting that S can be followed by non-terminals that correspond to 'b', 'c', 'd', 'e' and '*$*'

"**S → Ab**" allows S to start with 'a' and 'b' (matching some part of First(S)) and be followed by non-terminals that correspond to 'b', 'c', 'd', or 'e' (matching Follow(S)) as well as the end-of-input marker '$'.

Likewise,
First(A) includes {c, d, e, *ϵ*}, indicating that A can start with 'c', 'd, 'e', '*ϵ*'.
Follow(A) includes {b}, suggesting that A can be followed by non-terminals that correspond to 'b'.

The production "**A → XYZ**" allows A to start with 'ϵ', 'c', 'd', or 'e' (matching First(A)) and can be followed by 'b' (matching Follow(A)) using prod 1.
Choosing "**A → XYZ**" allows S to produce starting symbols like 'c', 'd', 'e' too which is making them the correct combination to choose.

4. Consider a lexical analyzer, which has been developed to recognize the tokens **alpha**, **beta** and **gamma**. The regular expressions for the aforesaid three tokens have been specified in the table below. The lexical analyzer was developed such that for a given input string, (a) it aims to recognize **all the** three tokens (maybe multiple times), if they are present in the string, (b) it prefers to match with the longest prefix with the respective regular expression.

Consider the input string **pppqppqprpqp**. Show the stream of tokens generated by the input string.

Regular Expression	**Token**
p*(p|q)p*	gamma
qp*	alpha
p+(r|q)pq	beta

Solution:

Logic: Catch the longest sequence

gamma = p*(p or q)p* -> **pppqpp** (possible string) / **p** (possible string)

alpha = qp* -> **q** (possible string)

beta = p+ (r or q) pq -> **prpq** (possible string)

pppqppqprpqp => pppqpp q prpq p -> longest continuous sequence

gamma alpha beta gamma (answer)