Section 4: CUP & LL

CSE 401/M501

Adapted from Autumn 2022

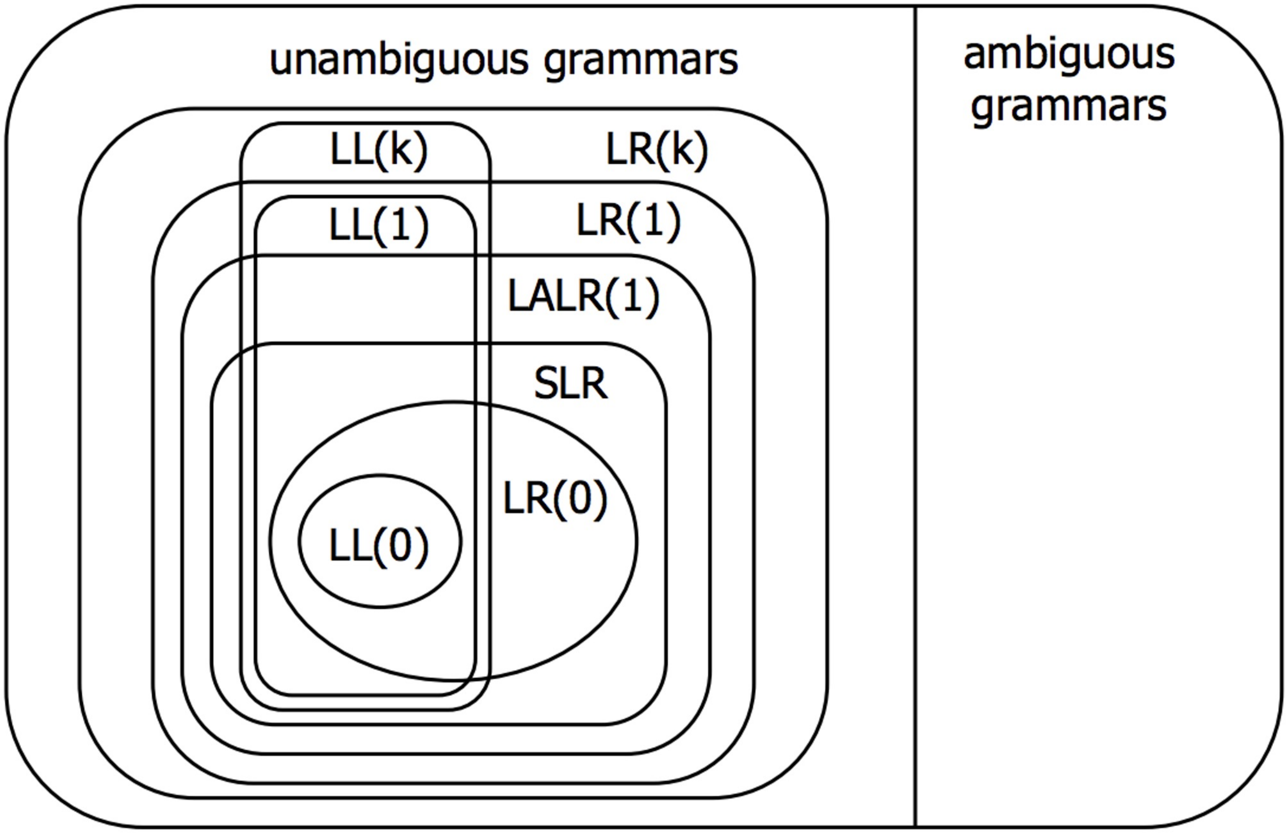
Administrivia

- Homework 2 is due tonight!
  - You have late days if you need them (2 max)
- Parser is due one week from today
  - Be sure to check your Scanner feedback – out later this week
- HW3 is out now as well, due in 1.5 weeks on ***Monday*** Oct. 30
  - **Only one late day allowed** on this assignment so we can distribute solutions before the midterm at the end of that week.

Parser Live Demo

A video recording of this demo will be posted on the website as a supplement

Language Hierarchies



The CUP parser generator

- Uses LALR(1)
  - A little weaker (less selective), but many fewer states than LR(1) parsers
  - Handles most realistic programming language grammars
  - More selective than SLR (or LR(0)) about when to do reductions, so works for more languages

The CUP parser generator

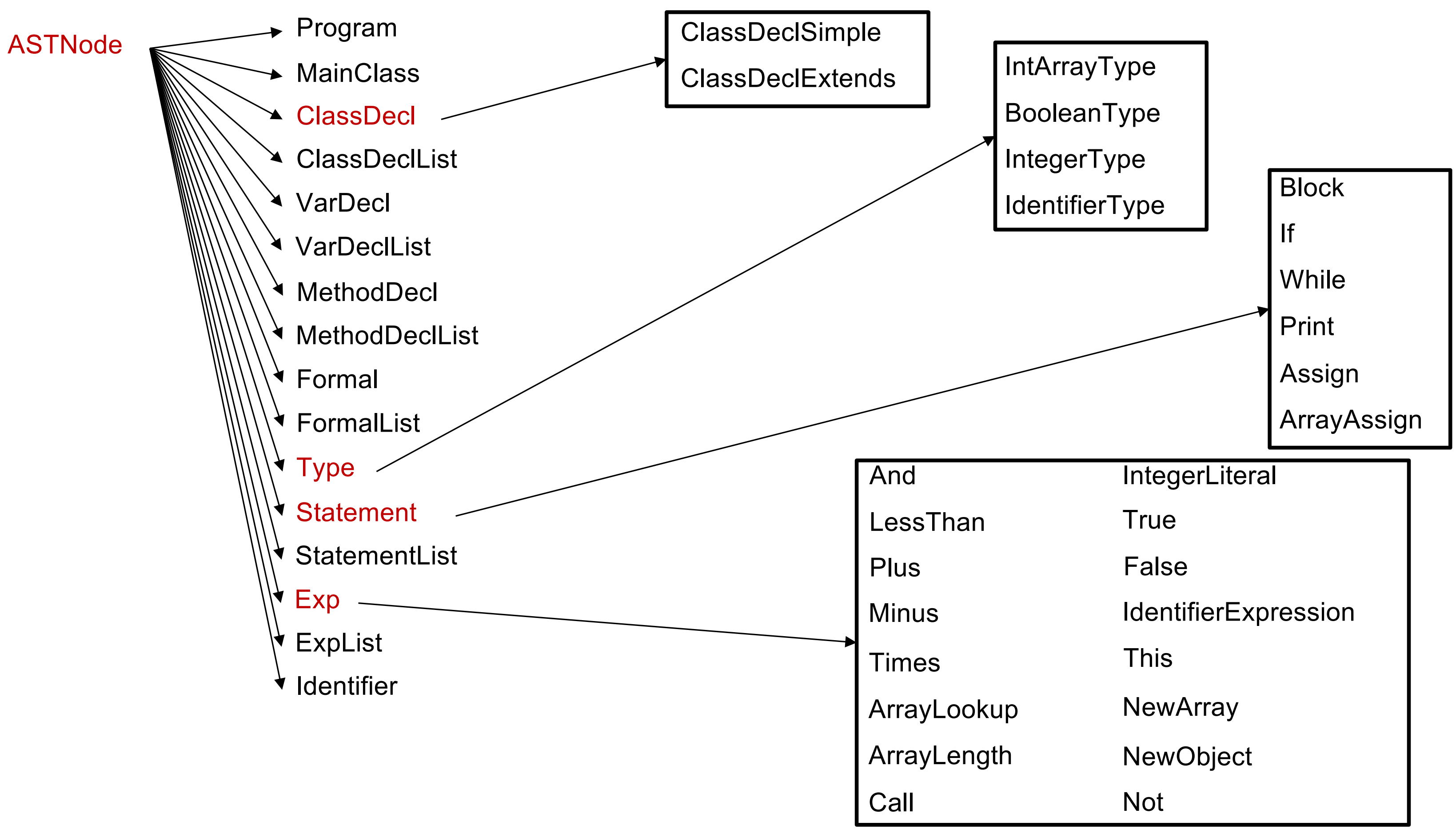
- LALR(1) parser generator based on YACC and Bison
- CUP can resolve some ambiguities itself
  - Precedence for reduce/reduce conflicts
  - Associativity for shift/reduce conflicts
  - Useful for our project for things like arithmetic expressions (exp+exp, exp*exp for fewer non-terminals, then add precedence and associativity declarations). ***Read the CUP docs!***

MiniJava BNF with AST Nodes

Use this to check your work ***only after*** your team has examined the grammar and AST code first.

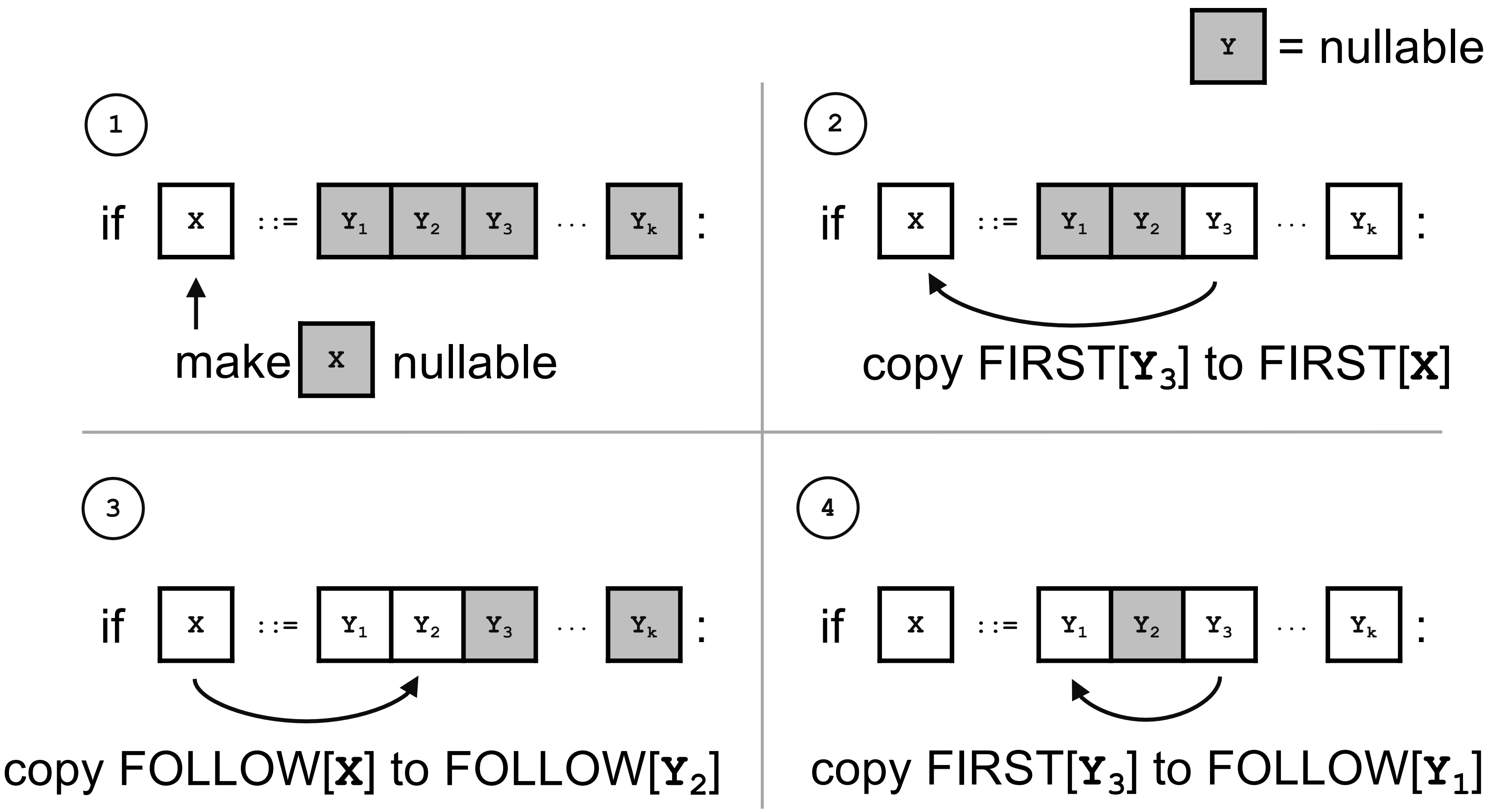
```
Program       Goal ::= MainClass ( ClassDeclaration )* <EOF>
MainClass     MainClass ::= "class" Identifier "{" "public" "static" "void" "main" "(" "String" "[" "]" Identifier ")" "{" Statement "}" "}"
ClassDecl     ClassDeclaration ::= "class" Identifier ( "extends" Identifier )? "{" ( VarDeclaration )* ( MethodDeclaration )* "}"   ClassDeclSimple
                                                                                                                                  ClassDeclExtends (if there is “extends”)
VarDecl       VarDeclaration ::= Type Identifier ";"   Formal
MethodDecl    MethodDeclaration ::= "public" Type Identifier "(" ( [Type Identifier] ( "," Type Identifier )* )? ")" "{" ( VarDeclaration )* ( Statement )* "return" Expression ";" "}"
                                                           (Formal → Type Identifier)
Type          Type ::= "int" "[" "]"          IntArrayType
                     | "boolean"              BooleanType
                     | "int"                  IntegerType
                     | Identifier             IdentifierType
Statement     Statement ::= "{" ( Statement )* "}"                               Block
                     | "if" "(" Expression ")" Statement "else" Statement       If
                     | "while" "(" Expression ")" Statement                     While
                     | "System.out.println" "(" Expression ")" ";"              Print
                     | Identifier "=" Expression ";"                            Assign
                     | Identifier "[" Expression "]" "=" Expression ";"         ArrayAssign
Exp           Expression ::= Expression ( "&&" | "<" | "+" | "-" | "*" ) Expression   And LessThan Plus Minus Times
                     | Expression "[" Expression "]"                            ArrayLookup
                     | Expression "." "length"                                  ArrayLength
                     | Expression "." Identifier "(" ( Expression ( "," Expression )* )? ")"   Call
                     | <INTEGER_LITERAL>                                        IntegerLiteral
                     | "true"                                                   True
                     | "false"                                                  False
                     | Identifier                                               IdentifierExpression
                     | "this"                                                   This
                     | "new" "int" "[" Expression "]"                           NewArray
                     | "new" Identifier "(" ")"                                 NewObject
                     | "!" Expression                                           Not
                     | "(" Expression ")"
Identifier    Identifier ::= <IDENTIFIER>
```

Abstract Syntax Tree Class Hierarchy



LL Parsing

Computing FIRST, FOLLOW, & nullable (3)



Computing FIRST, FOLLOW, and nullable

```
repeat
    for each production X := Y1 Y2 … Yk
        if Y1 … Yk are all nullable (or if k = 0)
           set nullable[X] = true
        for each i from 1 to k and each j from i +1 to k
           if Y1 … Yi-1 are all nullable (or if i = 1)
                  add FIRST[Yi] to FIRST[X]
           if Yi+1 … Yk are all nullable (or if i = k )
                  add FOLLOW[X] to FOLLOW[Yi]
           if Yi+1 … Yj-1 are all nullable (or if i+1=j)
                  add FIRST[Yj] to FOLLOW[Yi]
Until FIRST, FOLLOW, and nullable do not change
```

L L (k)

Left-to-Right
Only takes one pass, performed from the left

Leftmost
At each point, finds the derivation for the leftmost handle **(top-down)**

k Terminal Lookahead
Must determine derivation from the next unparsed terminal in the string

Typically *k* = 1, just like LR

LL(k) parsing

- LL(*k*) scans left-to-right, builds leftmost derivation, and looks ahead *k* symbols

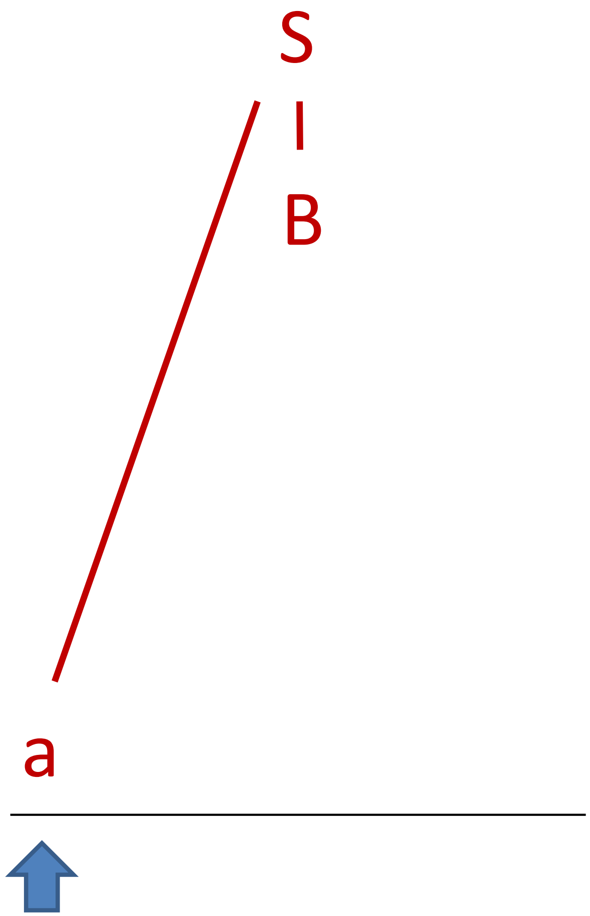
- The LL condition enable the parser to choose productions correctly with 1 symbol of look-ahead
- We can often transform a grammar to satisfy this if needed

LL(1) parsing: An example top-down derivation of "a z x"

```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | z
```

Lookahead	Remaining
a	z x

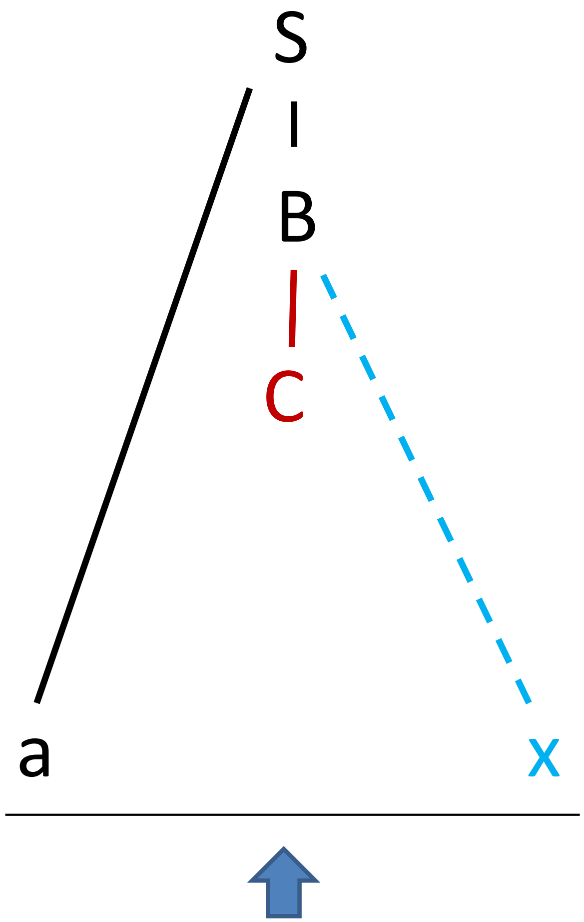
Top-Down Derivation of “a z x”




```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | z
```

Lookahead	Remaining
a	z x

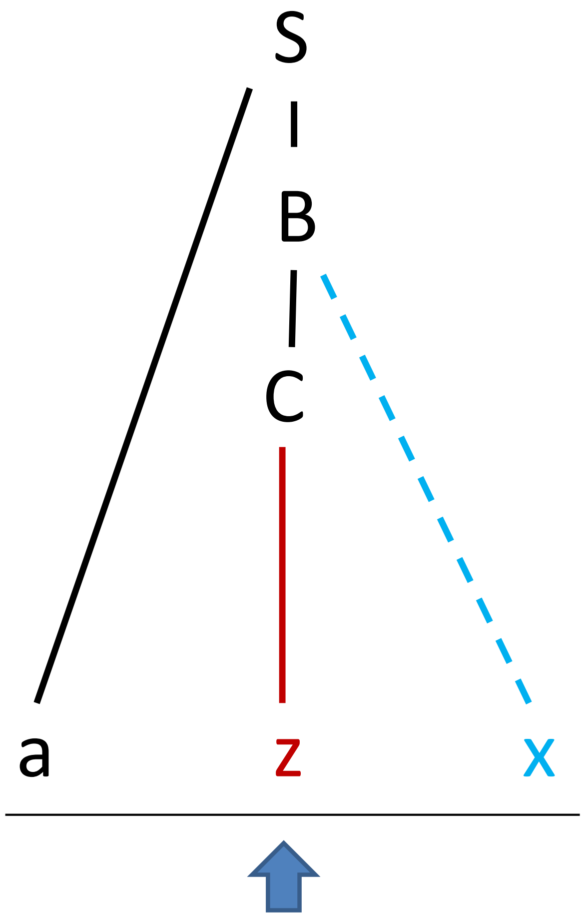
Top-Down Derivation of “a z x”




```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | z
```

Lookahead	Remaining
z	x

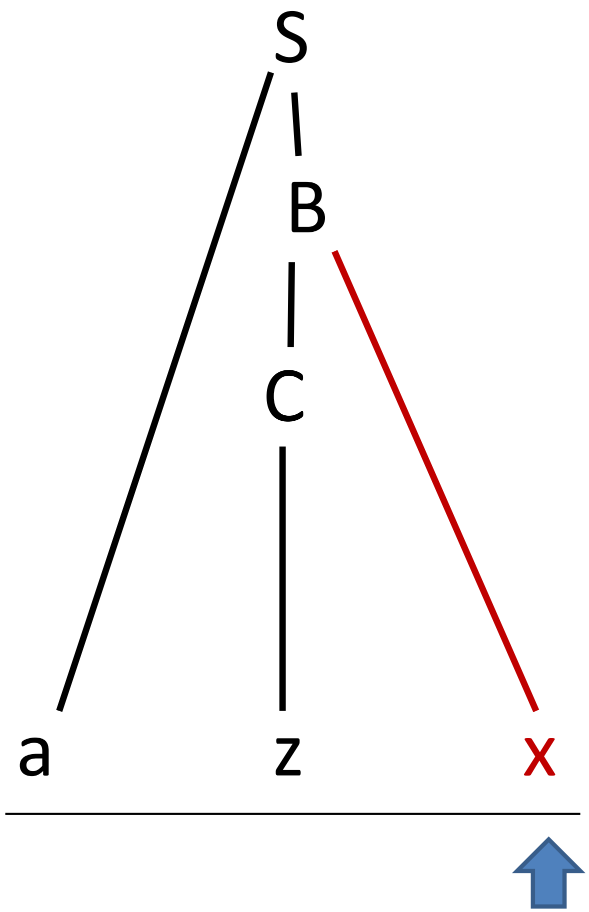
Top-Down Derivation of “a z x”



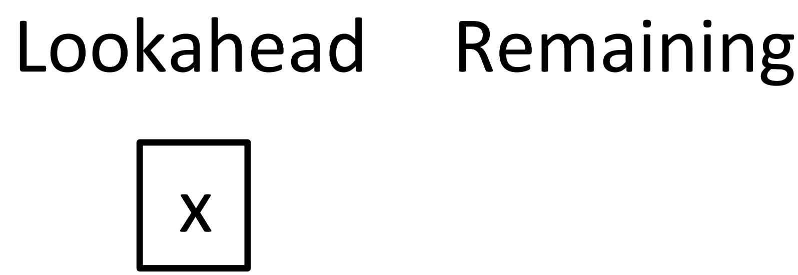

```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | z
```

Lookahead	Remaining
z	x

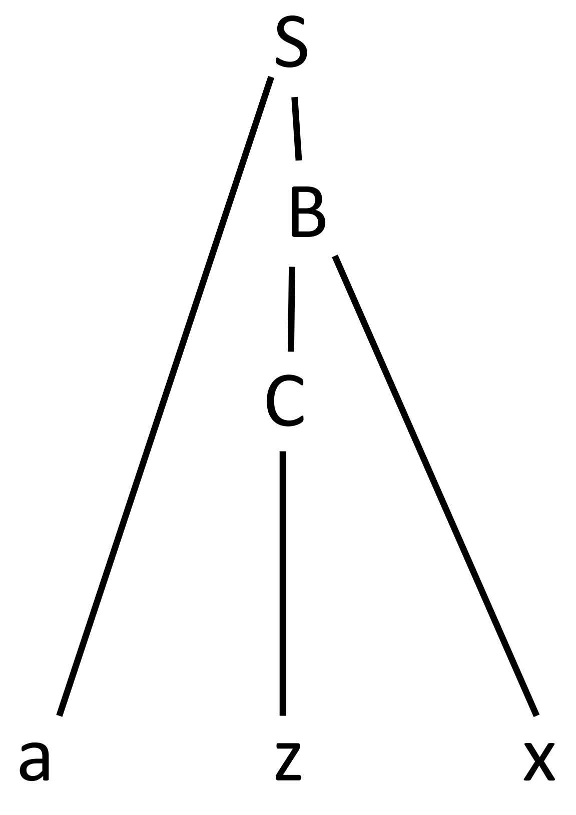
Top-Down Derivation of “a z x”




```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | z
```



Top-Down Derivation of “a z x”




```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | z
```

Successful parse!

LL Condition

For each nonterminal in the grammar:

- Its *productions* must have disjoint FIRST sets

✗
```
A ::= x | B
B ::= x
```

✓
```
A ::= x | B
B ::= y
```

- If it is *nullable*, the FIRST sets of its productions must be disjoint from its FOLLOW set

✗
```
S ::= A x
A ::= ε | x
```

✓
```
S ::= A y
A ::= ε | x
```

**We can often transform a grammar to satisfy this if needed

Canonical FIRST Conflict

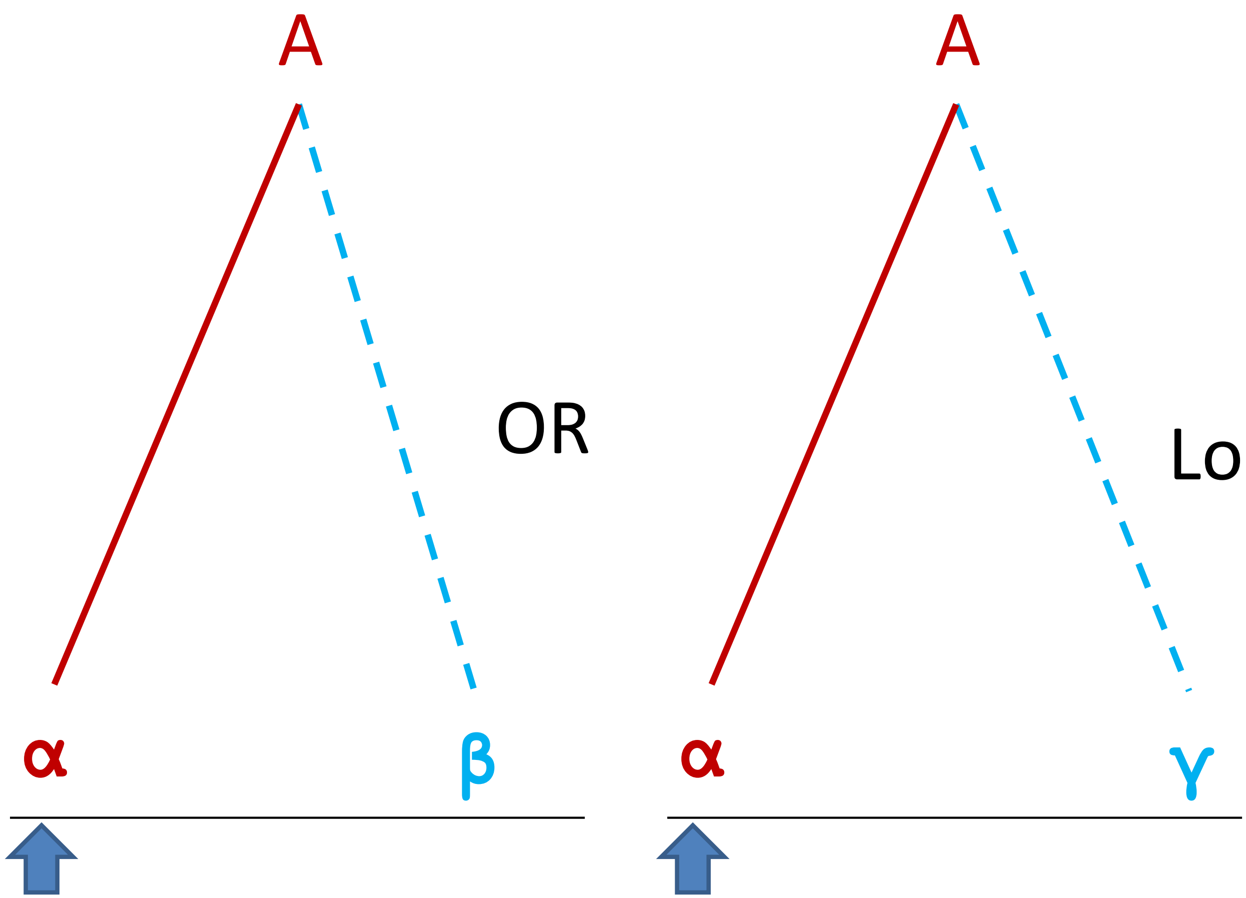
Problem

```
0. A ::= αβ | αγ
```

The FIRST sets of the right-hand sides for the SAME NON-TERMINAL must be disjoint!

Let's try a top-down derivation of αβ

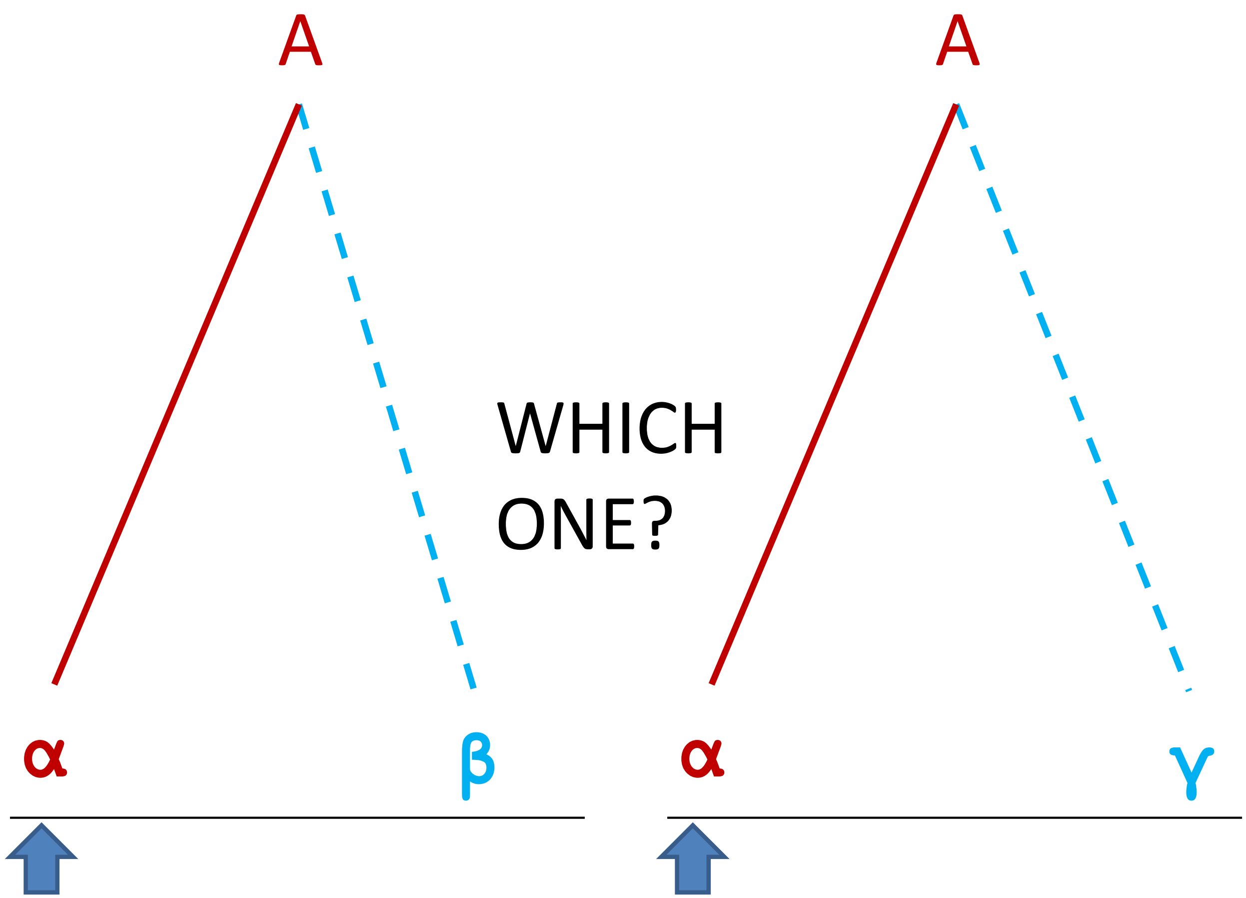
```
0. A ::= αβ | αγ
```



Lookahead	Remaining
α	β

Let's try a top-down derivation of αβ

```
0. A ::= αβ | αγ
```



We don't know!

We are using an LL(1) parser, we can't see more than α!

Canonical FIRST Conflict Solution

Solution

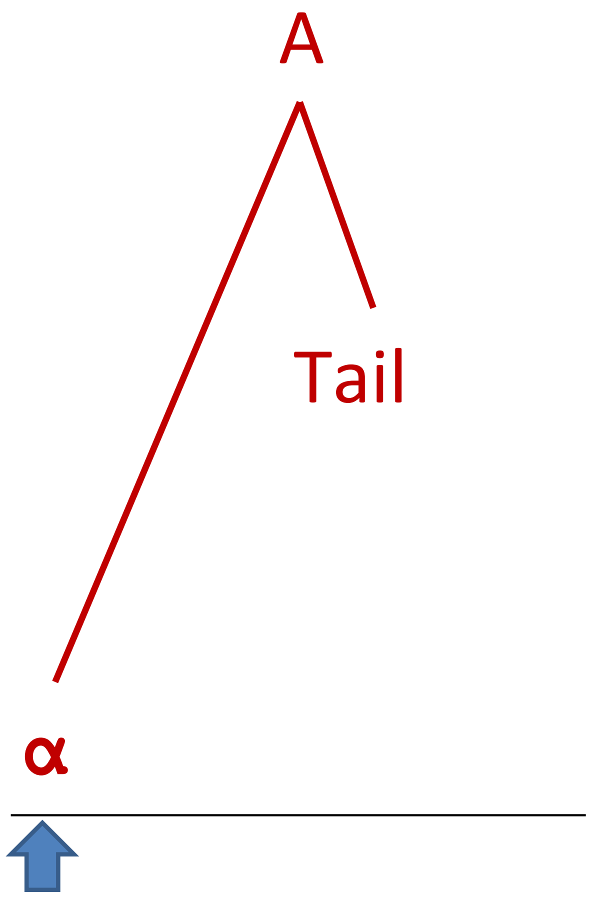
```
0. A ::= αβ | αγ

0. A ::= α Tail
1. Tail ::= β | γ
```

Factor out the common prefix

When multiple productions of a nonterminal share a common prefix, turn the different suffixes into a new nonterminal.

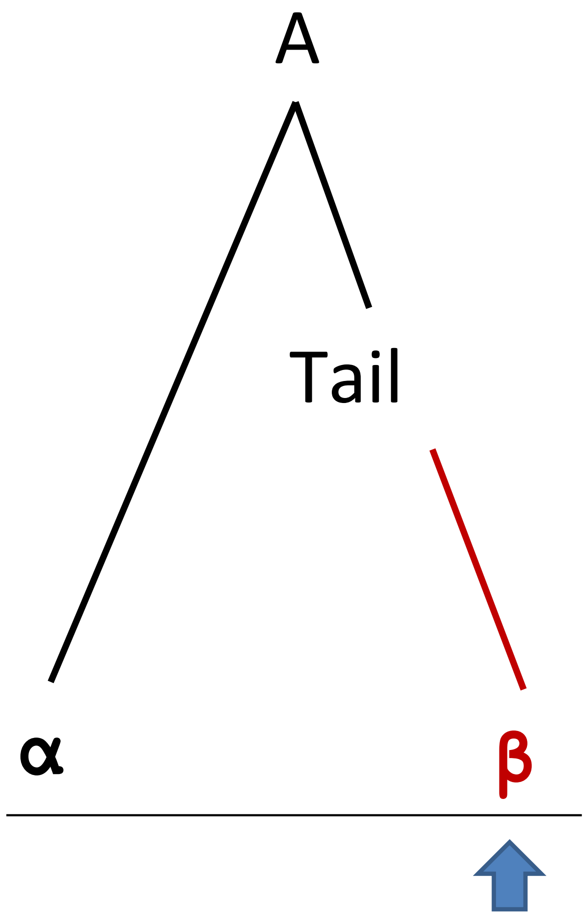
Top-Down Derivation of “αβ”




```
0. A ::= α Tail
1. Tail ::= β | γ
```

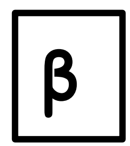
Lookahead	Remaining
α	β

Top-Down Derivation of “αβ”

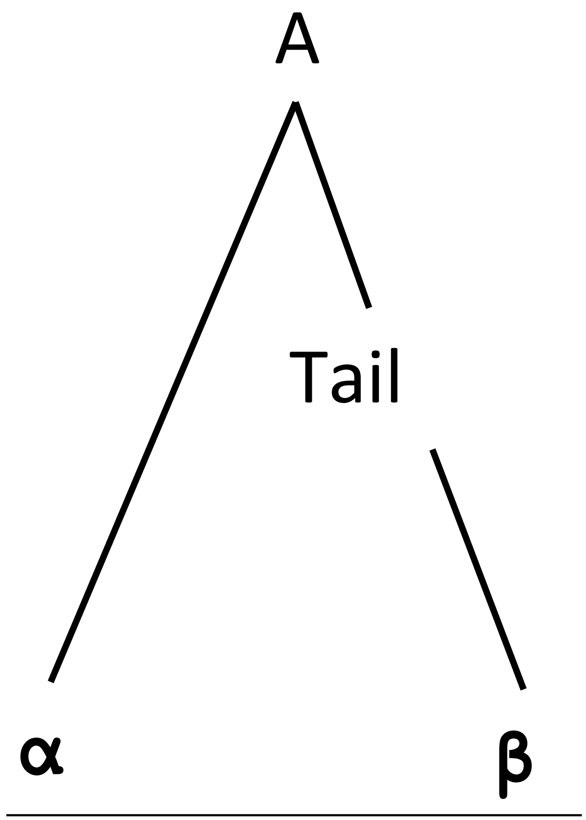



```
0. A ::= α Tail
1. Tail ::= β | γ
```

Lookahead Remaining



Top-Down Derivation of "αβ"




```
0. A ::= α Tail
1. Tail ::= β | γ
```

Successful parse!

Changing original grammar a little (Grammar 1)

```
0. S ::= a B | a w
1. B ::= C x | y
2. C ::= ε | z
```

Lookahead	Remaining
a	z x

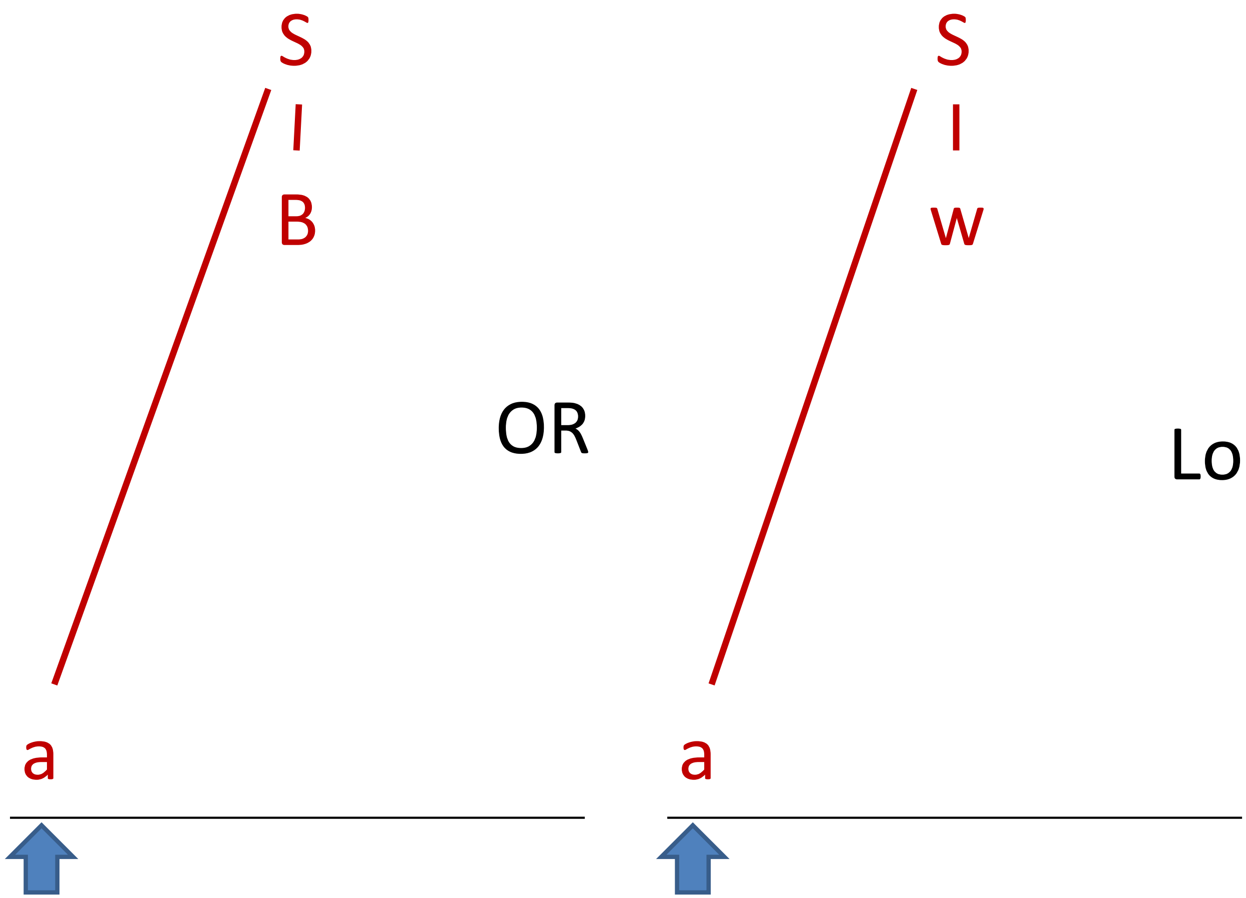
What's the issue?

```
0. S ::= a B | a w
1. B ::= C x | y
2. C ::= ε | z
```

There's a FIRST Conflict!

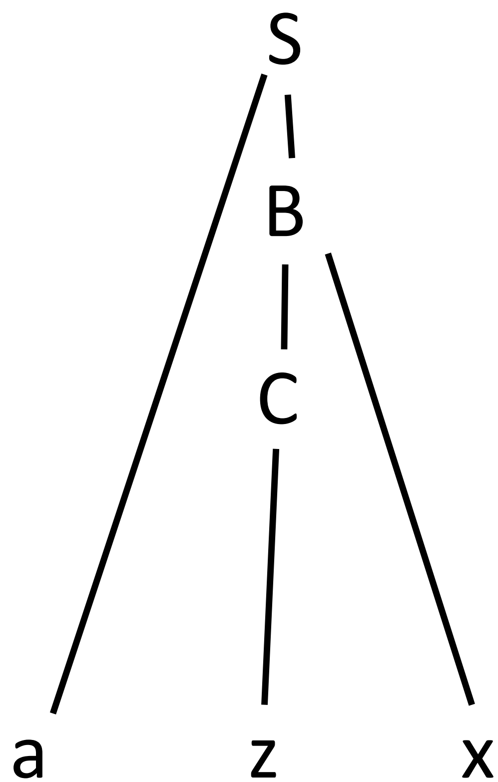
Top-Down Derivation of “a z x”: LL(1) can’t parse

```
0. S ::= a B | a w
1. B ::= C x | y
2. C ::= ε | z
```



Lookahead	Remaining
a	z x

Parse Tree without changing Grammar



0. S ::= a B | a w

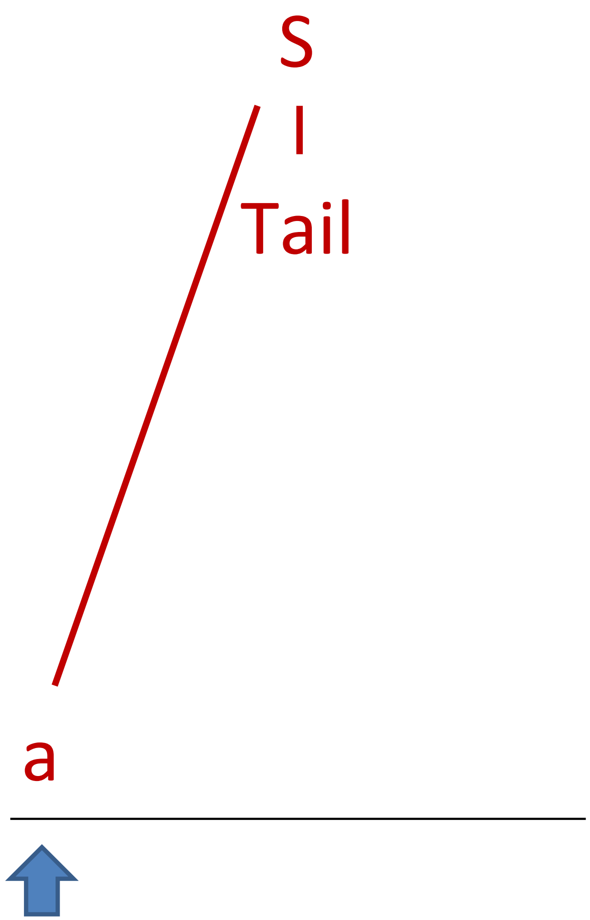
1. B ::= C x | y

2. C ::= ε | z

Applying the Fix: Factor out the Common Prefix

```
0. S ::= a Tail
1. Tail ::= B | w
2. B ::= C x | y
3. C ::= ε | z
```

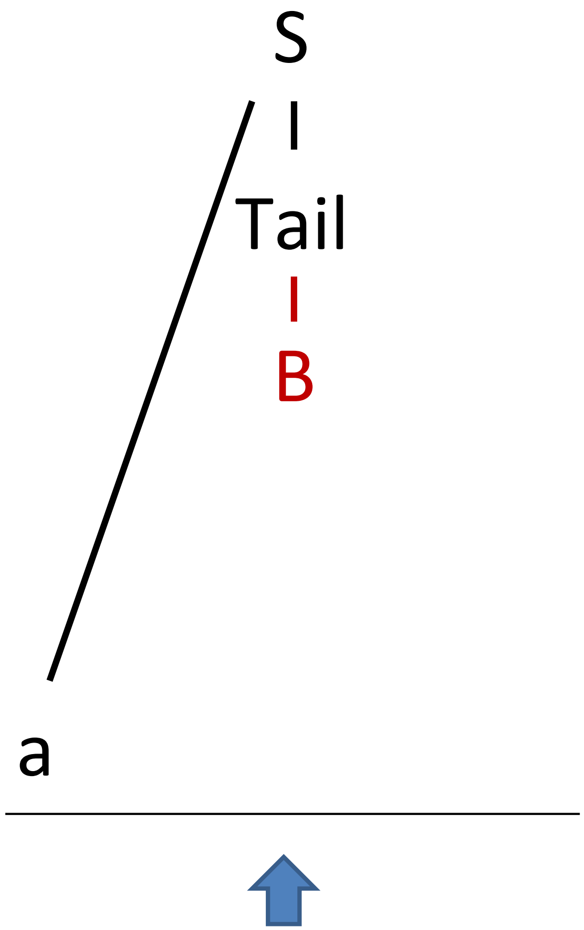
Top-Down Derivation of “a z x”




```
0. S ::= a Tail
1. Tail ::= B | w
2. B ::= C x | y
3. C ::= ε | z
```

Lookahead	Remaining
a	z x

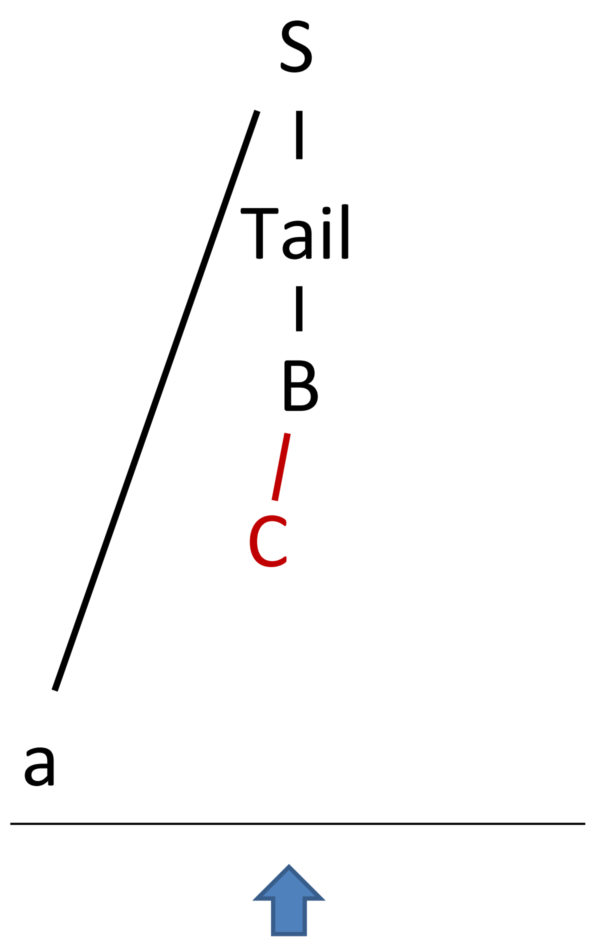
Top-Down Derivation of “a z x”




```
0. S ::= a Tail
1. Tail ::= B | w
2. B ::= C x | y
3. C ::= ε | z
```

Lookahead	Remaining
z	x

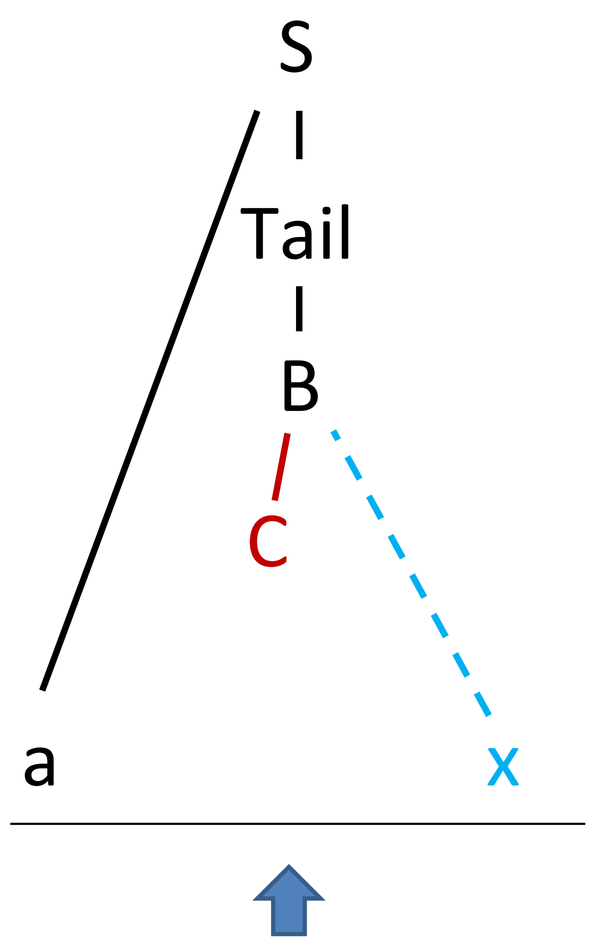
Top-Down Derivation of “a z x”




```
0. S ::= a Tail
1. Tail ::= B | w
2. B ::= C x | y
3. C ::= ε | z
```

Lookahead	Remaining
z	x

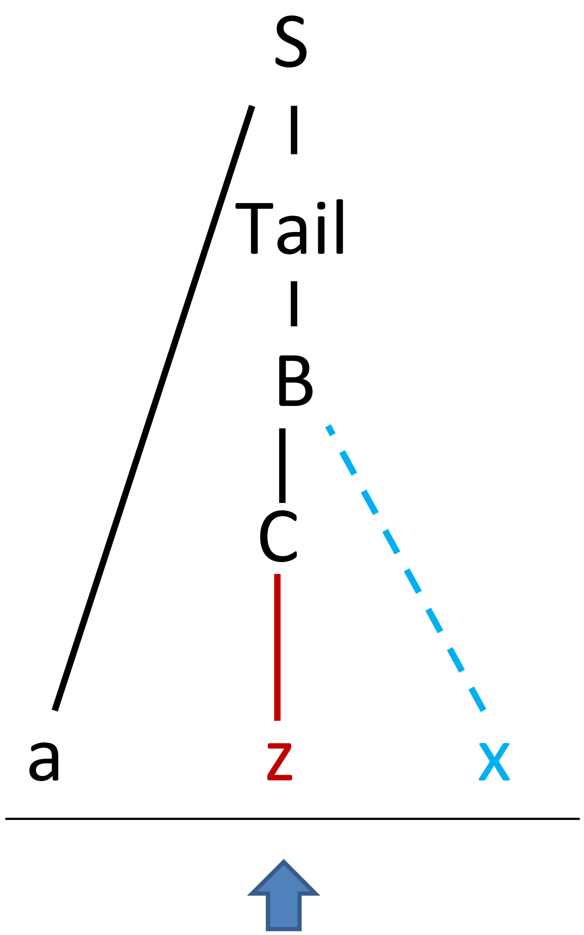
Top-Down Derivation of “a z x”




```
0. S ::= a Tail
1. Tail ::= B | w
2. B ::= C x | y
3. C ::= ε | z
```

Lookahead	Remaining
z	x

Top-Down Derivation of “a z x”



0. S ::= a Tail

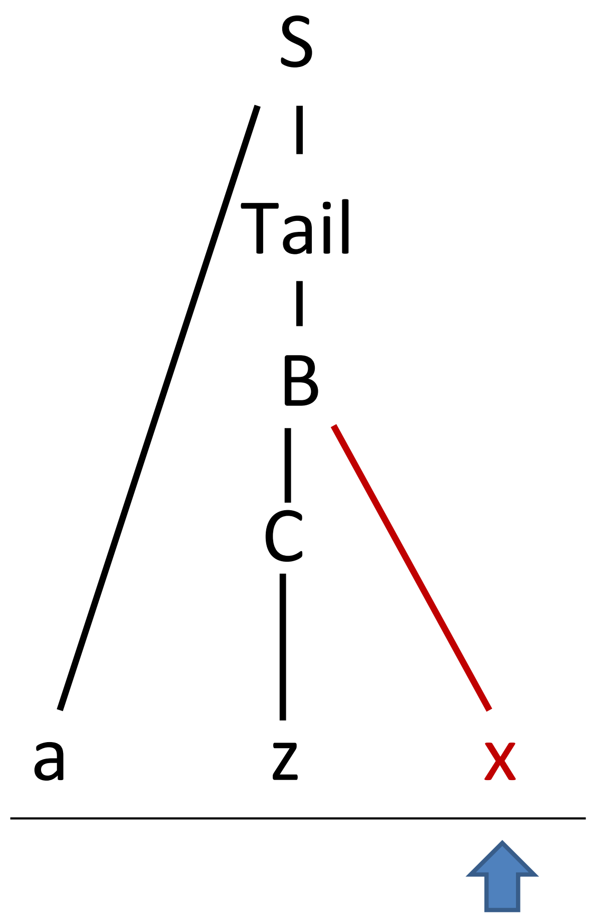
1. Tail ::= B | w

2. B ::= C x | y

3. C ::= ε | z

Lookahead	Remaining
z	x

Top-Down Derivation of “a z x”

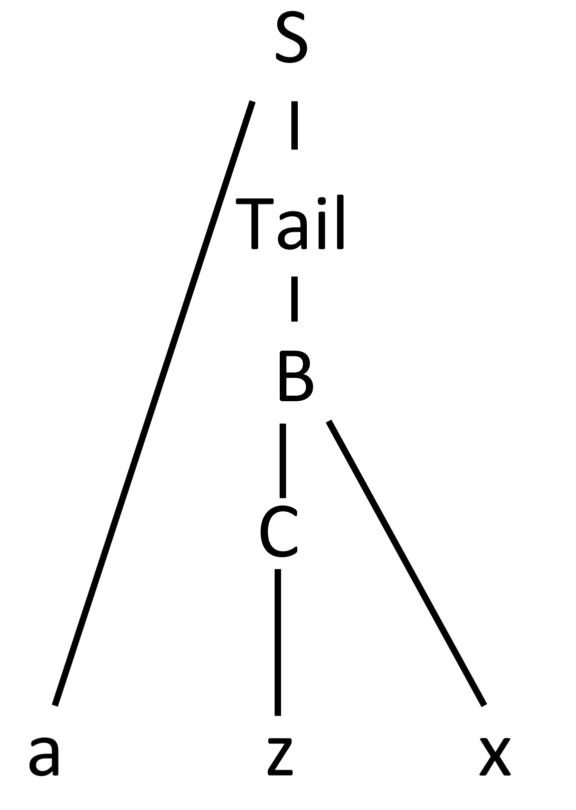



```
0. S ::= a Tail
1. Tail ::= B | w
2. B ::= C x | y
3. C ::= ε | z
```

Lookahead Remaining

Lookahead	Remaining
x	

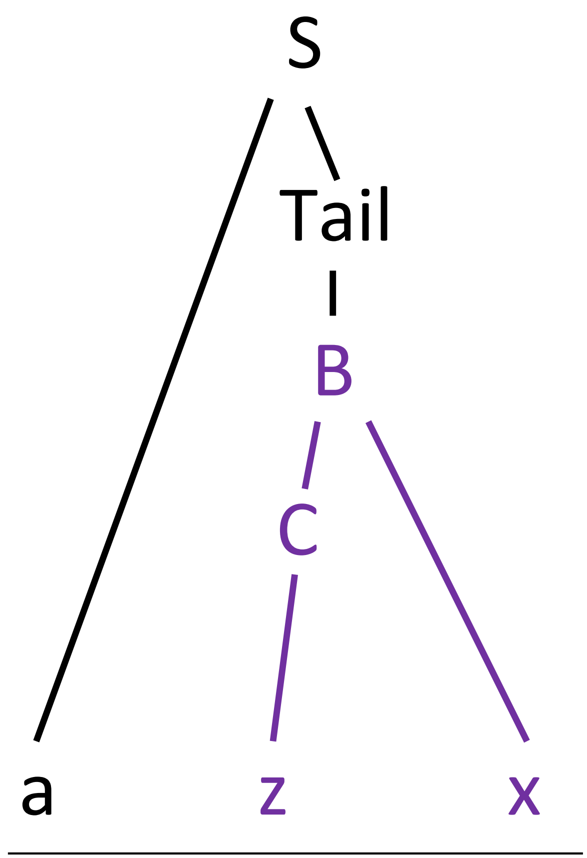
Top-Down Derivation of “a z x”



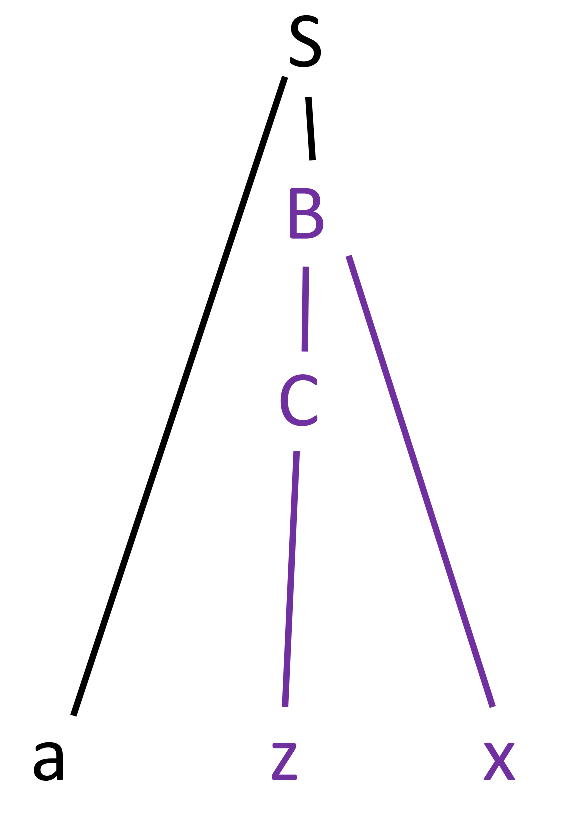

```
0. S ::= a Tail
1. Tail ::= B | w
2. B ::= C x | y
3. C ::= ε | z
```

Success!

Comparing Parse Trees



Purple trees are the same!



LL Condition

For each nonterminal in the grammar:

- Its *productions* must have disjoint FIRST sets

✗
```
A ::= x | B
B ::= x
```

✓
```
A ::= x | B
B ::= y
```

- If it is *nullable*, the FIRST sets of its productions must be disjoint from its FOLLOW set

✗
```
S ::= A x
A ::= ε | x
```

✓
```
S ::= A y
A ::= ε | x
```

**We can often transform a grammar to satisfy this if needed

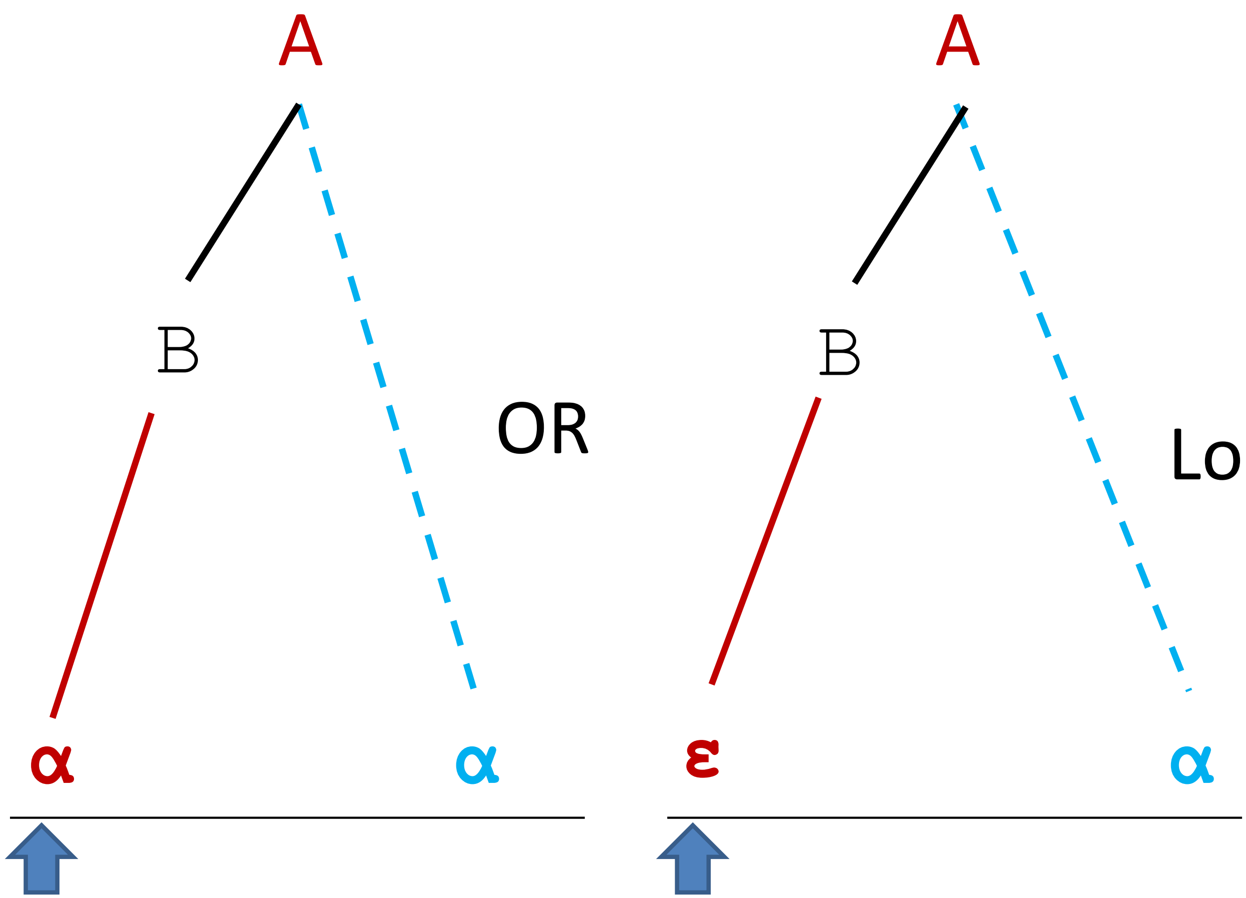
Canonical FIRST FOLLOW Conflict

Problem

```
0. A ::= B α
1. B ::= α | ε
```

Because B is nullable, its FOLLOW set must be disjoint from the FIRST sets of its right-hand sides!

Let’s try a top-down derivation of “α”

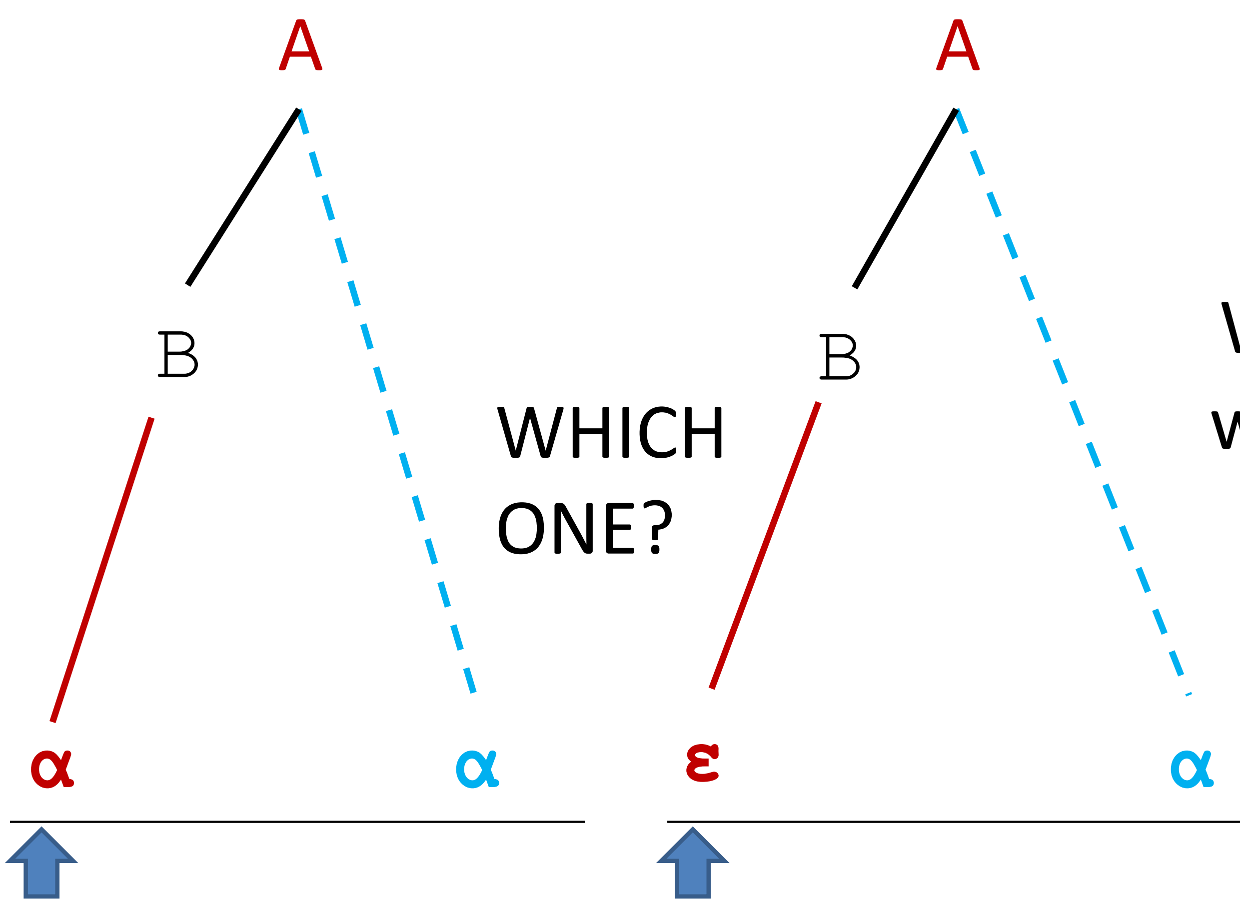



```
0. A ::= B α
1. B ::= α | ε
```

Lookahead Remaining

α

Let’s try a top-down derivation of “α”




```
0. A ::= B α
1. B ::= α | ε
```

We don’t know! Again, we can’t see more than α!

Canonical FIRST FOLLOW Conflict Solution

Solution

```
0. A ::= B α
1. B ::= α | ε
```

Substitute the common prefix

```
0. A ::= αα | α
```

Factor out the tail

```
0. A ::= α Tail
1. Tail ::= α | ε
```

Watch out for Nullability! (Grammar 2)

Changing the grammar again…

```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | x
```

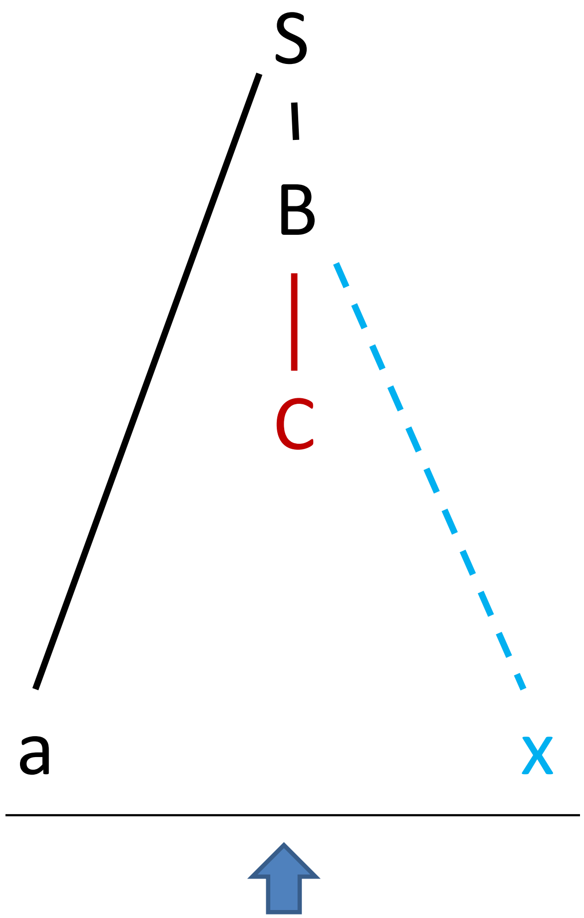
Lookahead	Remaining
a	x

What's the issue?

```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | x
```

FIRST FOLLOW Conflict

Top down derivation of “ax”



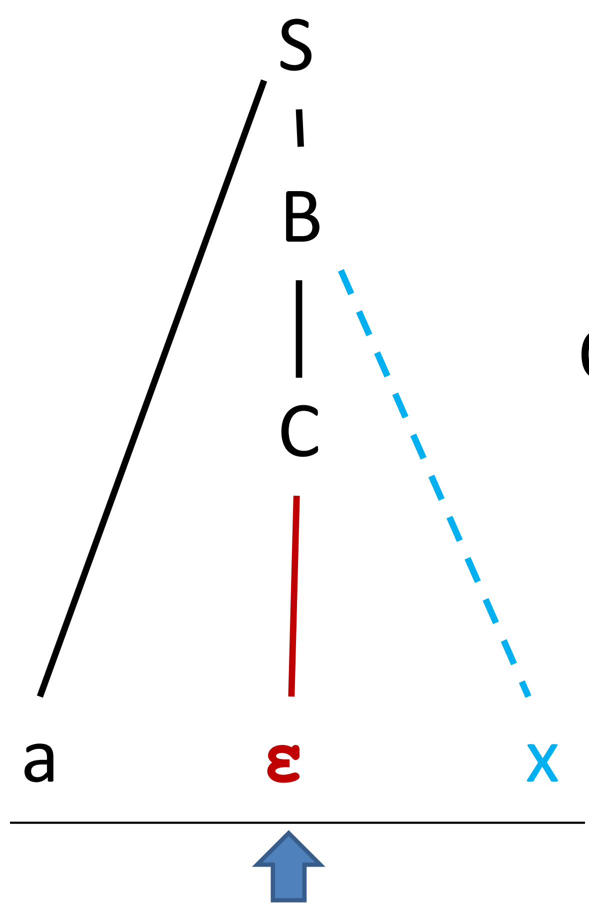

```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | x
```

Lookahead Remaining

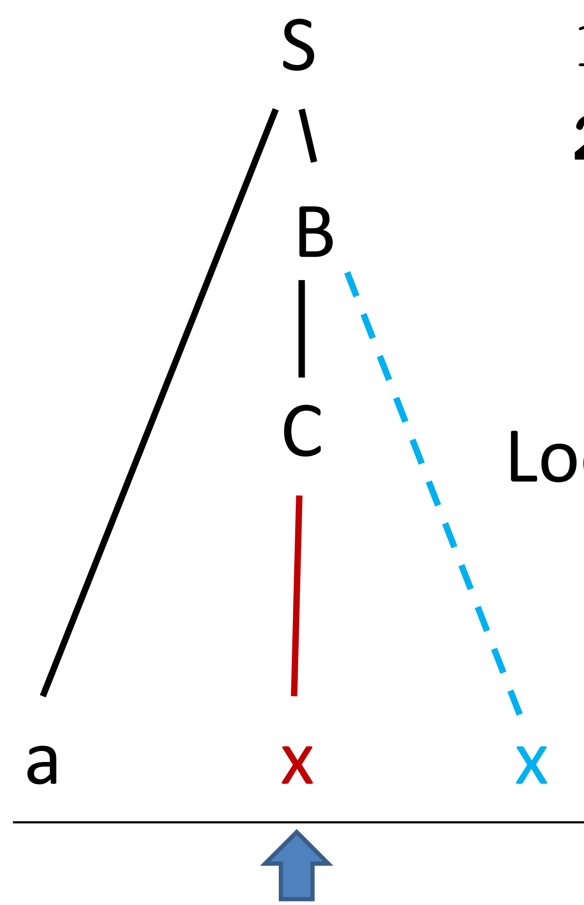
x

Top down derivation of "ax"

```
0. S ::= a B
1. B ::= C x | y
```
`2. C ::= ε | x`



OR



Lookahead Remaining

x

Applying the Fix: Substitute the Common Prefix,

1

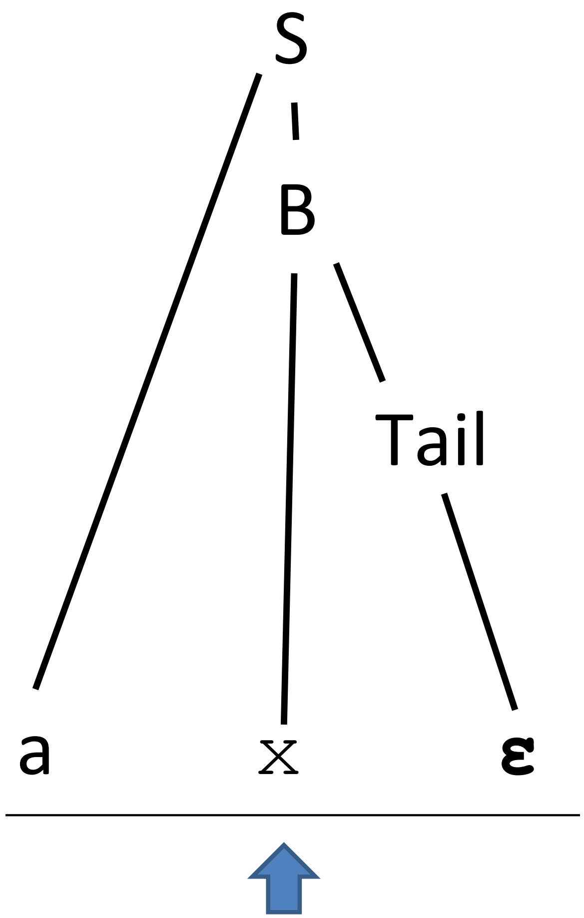
```
0. S ::= a B
1. B ::= x | xx | y
```

~~2. C ::= ε | x~~

2

```
0. S ::= a B
1. B ::= x Tail | y
2. Tail ::= x | ε
```

Top down derivation of “ax”




```
0. S ::= a B
1. B ::= x Tail | y
2. Tail ::= x | ε
```