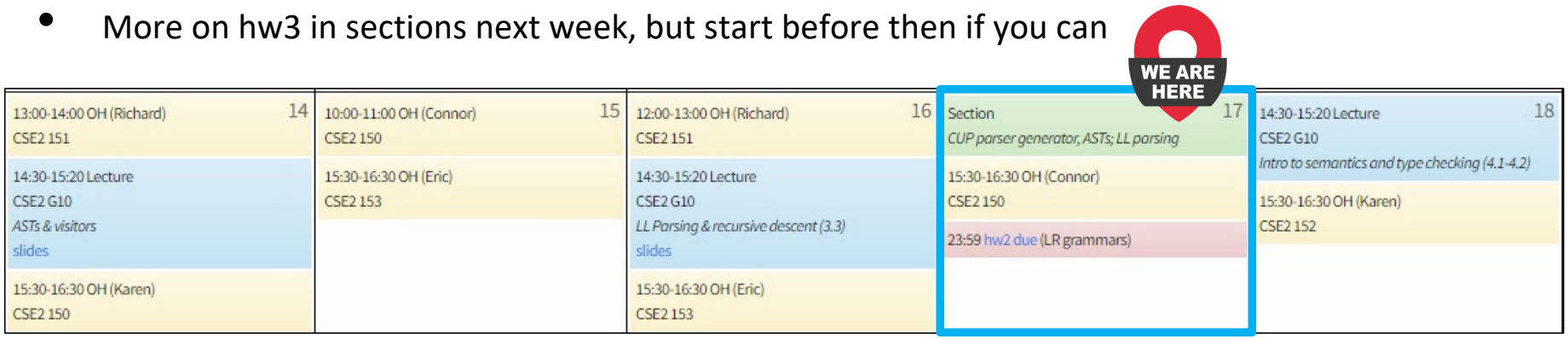
Section 4: CUP & LL

CSE 401/M501

Adapted from Autumn 2022

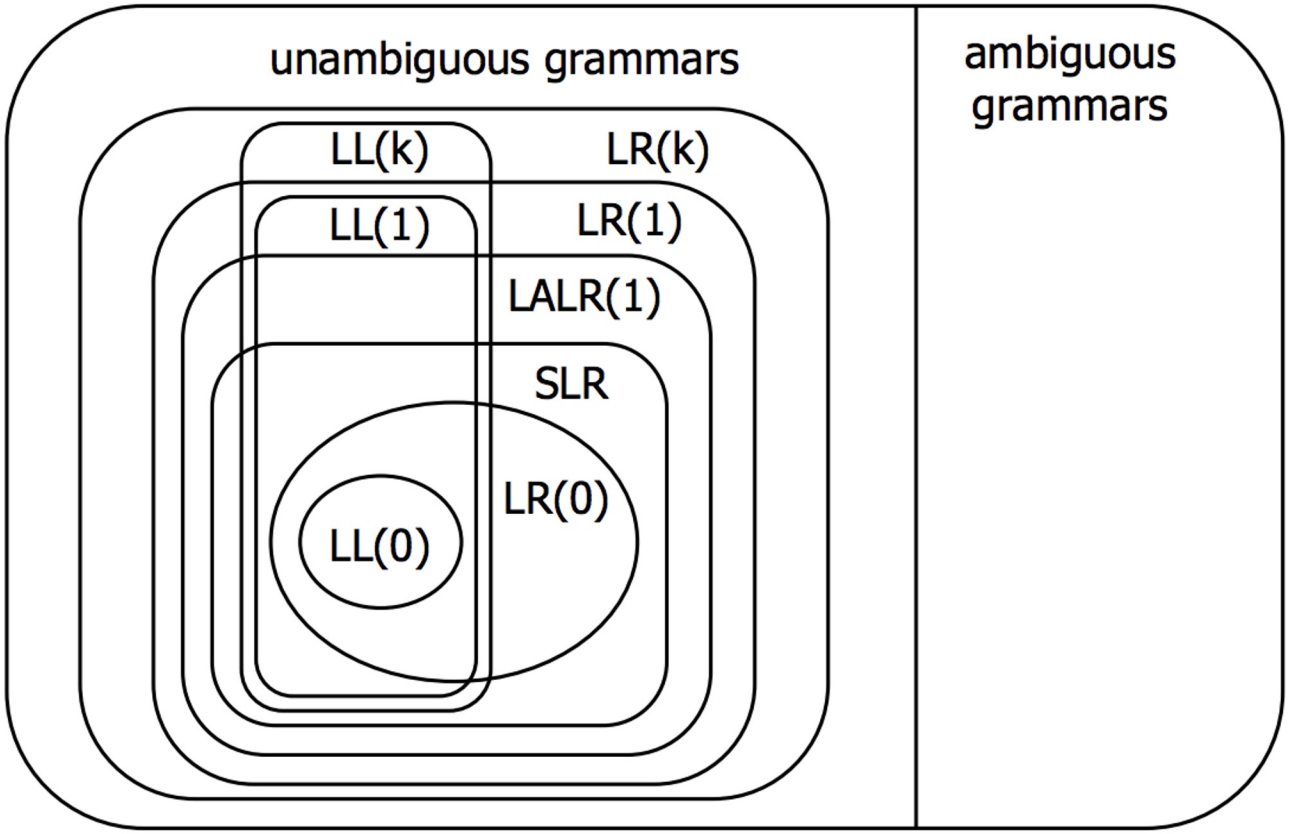
Administrivia

- 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 tomorrow, due in 1.5 weeks on ***Monday, October 28th***
  - **Only one late day allowed** on this assignment so we can distribute solutions before the midterm at the end of that week.
  - More on hw3 in sections next week, but start before then if you can



Parser Live Demo

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 (use exp+exp, exp*exp, etc. for fewer non-terminals, then add precedence and associativity declarations). ***Read the CUP docs!***

MiniJava Grammar -> AST Node

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

Program
MainClass
ClassDecl
VarDecl
MethodDecl
Type
Statement
Exp
Identifier

```
Goal ::= MainClass ( ClassDeclaration )* <EOF>
MainClass ::= "class" Identifier "{" "public" "static" "void" "main" "(" "String" "[" "]" Identifier ")" "{" Statement "}" "}"
ClassDeclaration ::= "class" Identifier ( "extends" Identifier )? "{" ( VarDeclaration )* ( MethodDeclaration )* "}"
VarDeclaration ::= Type Identifier ";"
MethodDeclaration ::= "public" Type Identifier "(" ( Type Identifier ( "," Type Identifier )* )? ")" "{" ( VarDeclaration )* ( Statement )* "return" Expression ";" "}"
Type ::= "int" "[" "]"
       | "boolean"
       | "int"
       | Identifier
Statement ::= "{" ( Statement )* "}"
       | "if" "(" Expression ")" Statement "else" Statement
       | "while" "(" Expression ")" Statement
       | "System.out.println" "(" Expression ")" ";"
       | Identifier "=" Expression ";"
       | Identifier "[" Expression "]" "=" Expression ";"
Expression ::= Expression ( "&&" | "<" | "+" | "-" | "*" ) Expression
       | Expression "[" Expression "]"
       | Expression "." "length"
       | Expression "." Identifier "(" ( Expression ( "," Expression )* )? ")"
       | <INTEGER_LITERAL>
       | "true"
       | "false"
       | Identifier
       | "this"
       | "new" "int" "[" Expression "]"
       | "new" Identifier "(" ")"
       | "!" Expression
       | "(" Expression ")"
Identifier ::= <IDENTIFIER>
```

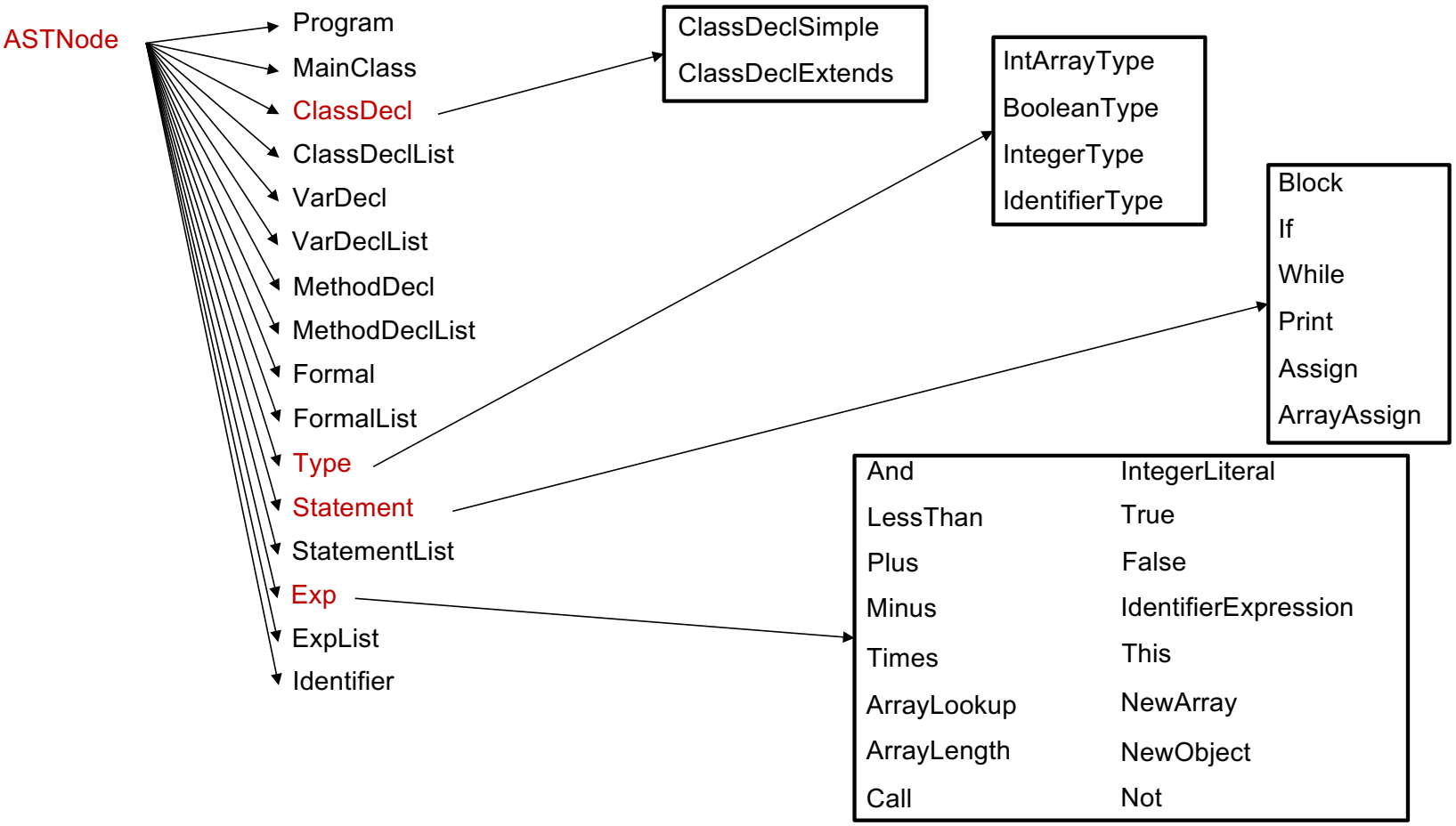
Formal

Block

ClassDeclSimple

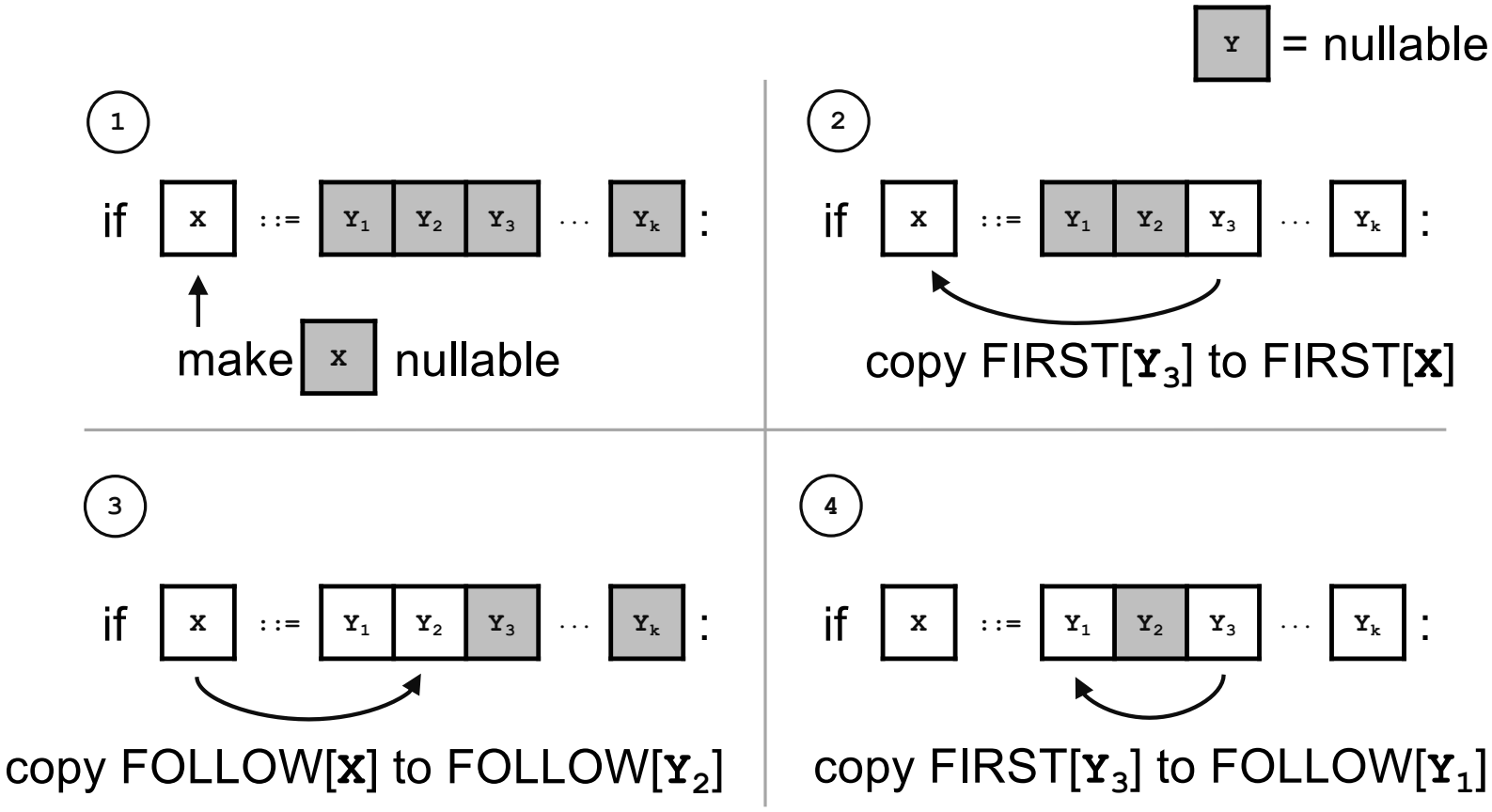
ClassDeclExtends (if there is “extends”)

Abstract Syntax Tree Class Hierarchy



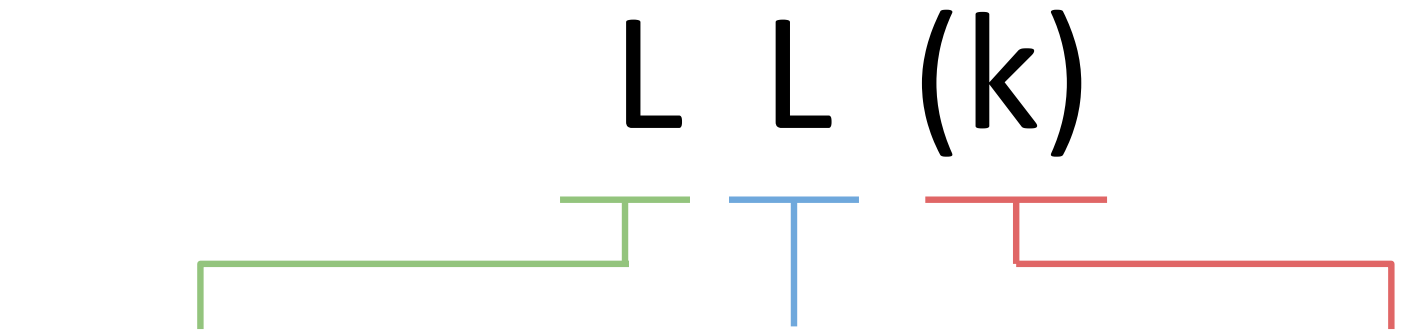
LL Parsing

Computing FIRST, FOLLOW, & nullable (3)



Computing FIRST, FOLLOW, and nullable

repeat
  for each production $X := Y_1\ Y_2 \ldots Y_k$
    if $Y_1 \ldots Y_k$ 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 $Y_1 \ldots Y_{i-1}$ are all nullable (or if $i = 1$)
        add FIRST[Y_i] to FIRST[X]
      if $Y_{i+1} \ldots Y_k$ are all nullable (or if $i = k$)
        add FOLLOW[X] to FOLLOW[Y_i]
      if $Y_{i+1} \ldots Y_{j-1}$ are all nullable (or if i+1=j)
        add FIRST[Y_j] to FOLLOW[Y_i]
Until FIRST, FOLLOW, and nullable do not change



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

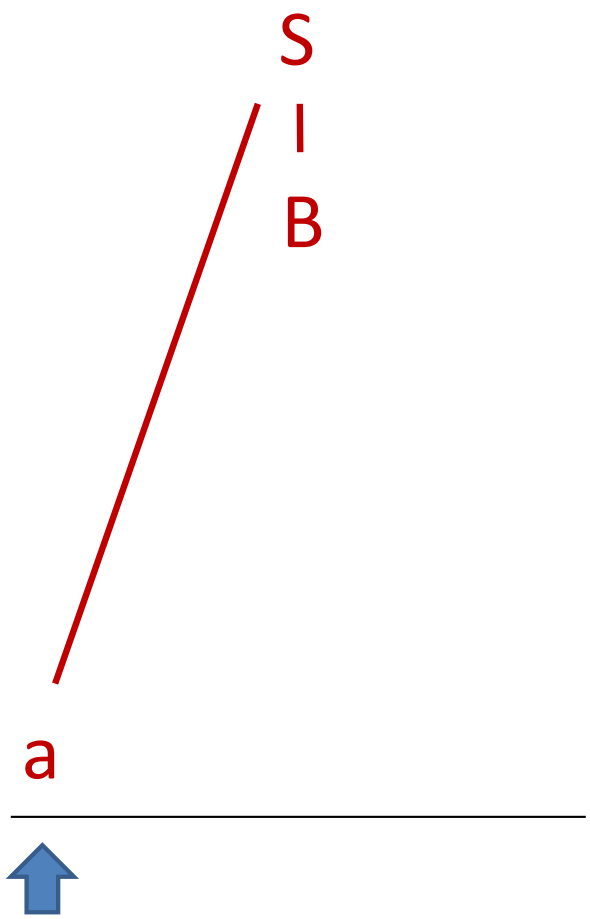
- The LL condition enable the parser to choose productions correctly with 1 symbol of look-ahead
- We can often transform a grammar to satisfy this if needed

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

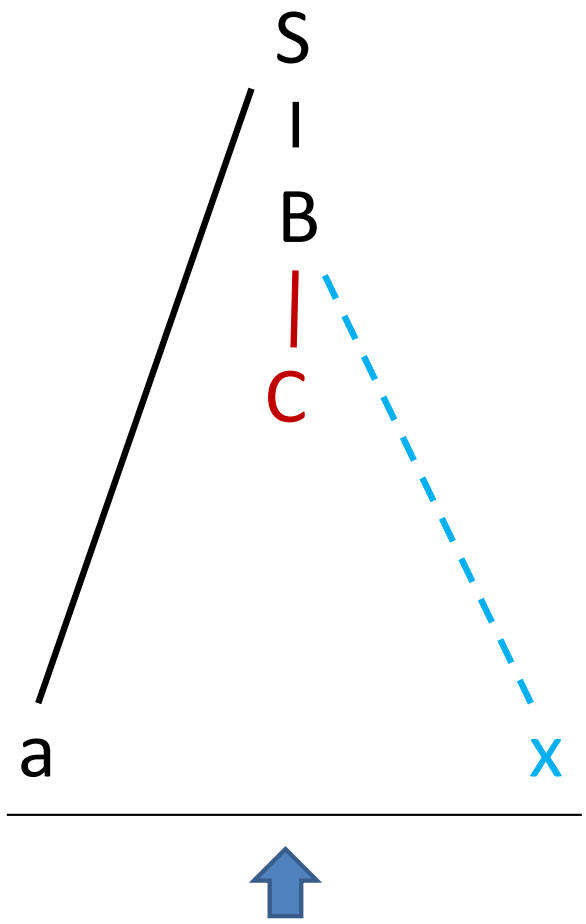
Top-Down Derivation of “a z x”




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

Lookahead	Remaining
a	z x

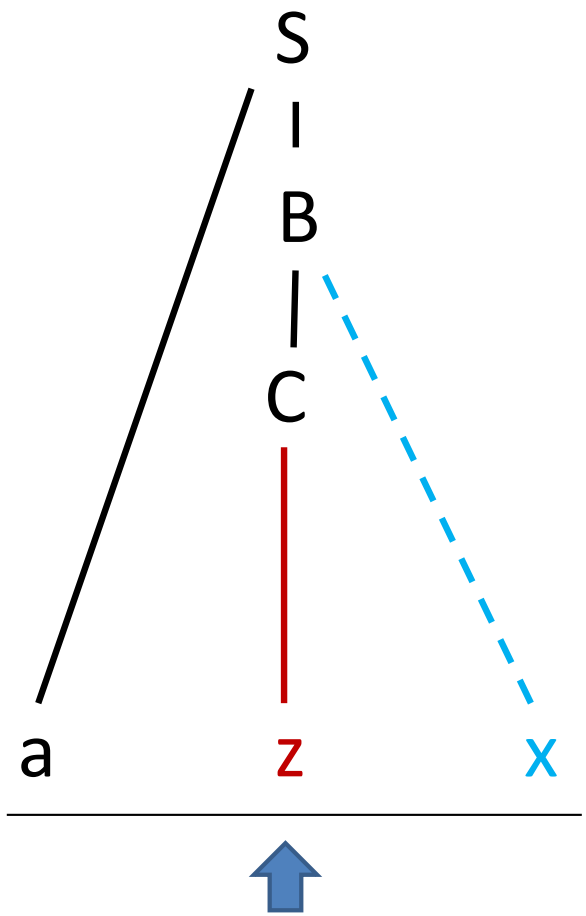
Top-Down Derivation of “a z x”




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

Lookahead	Remaining
z	x

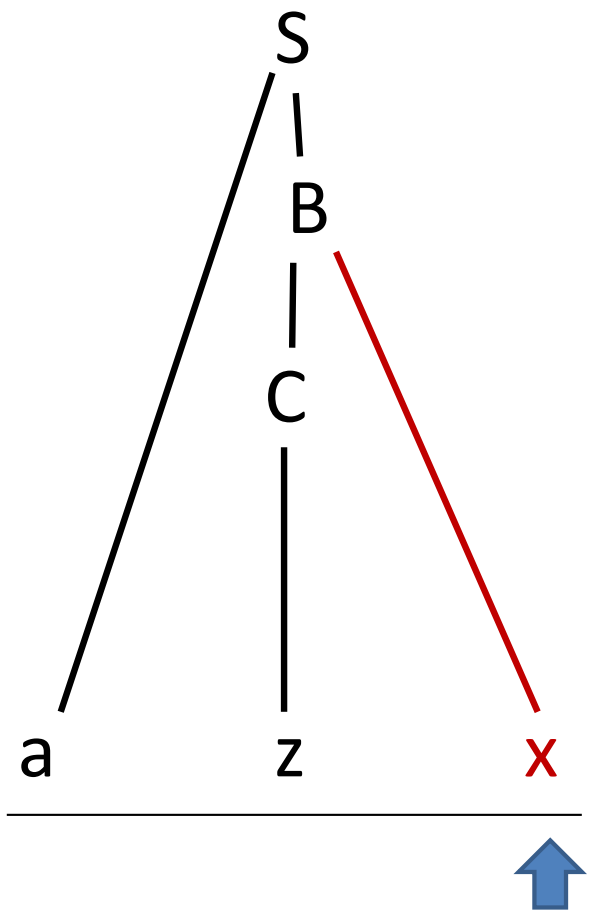
Top-Down Derivation of “a z x”




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

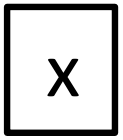
Lookahead	Remaining
z	x

Top-Down Derivation of “a z x”

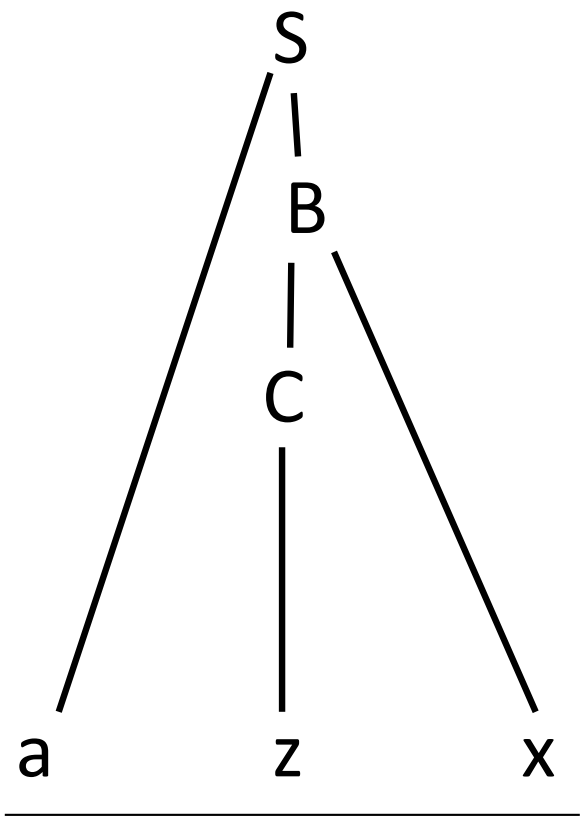



```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | z
```

Lookahead Remaining



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

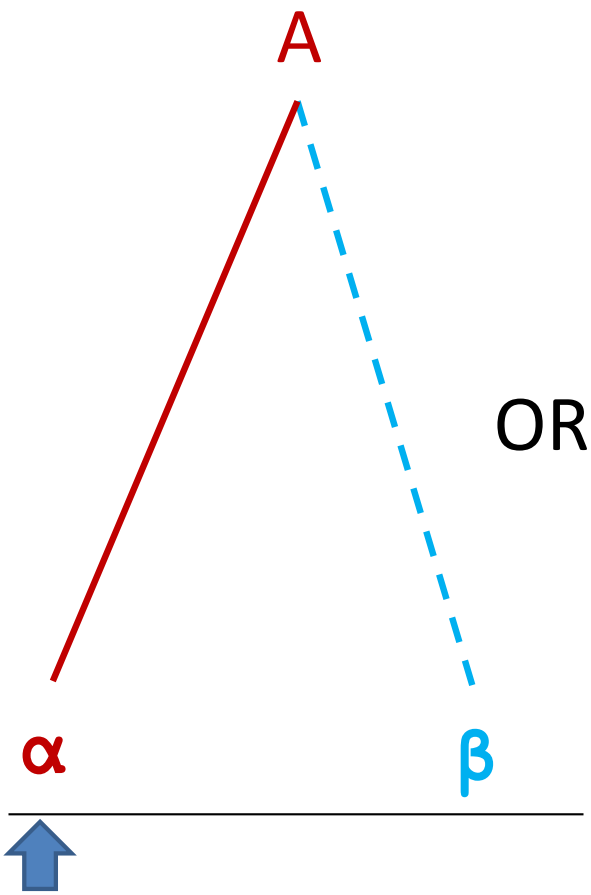
Problem

0. **A** ::= αβ | αγ

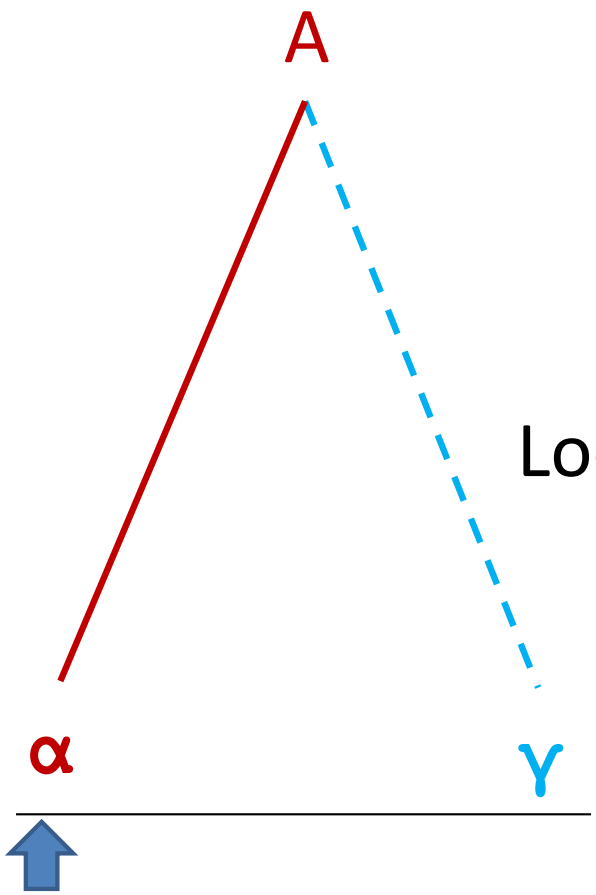
The FIRST sets of the right-hand sides for the SAME NON-TERMINAL must be disjoint!

Let's try a top-down derivation of αβ

0. A ::= αβ | αγ



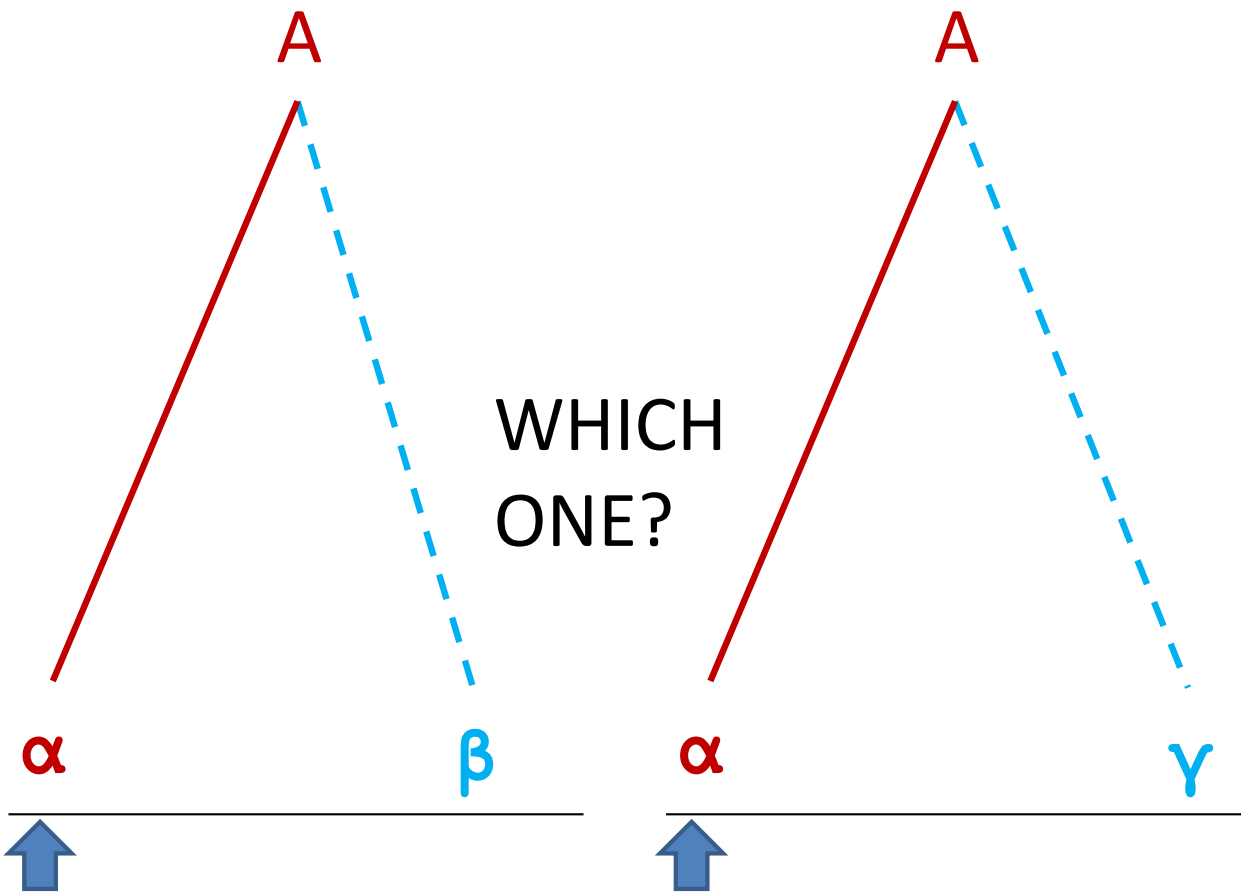
OR



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

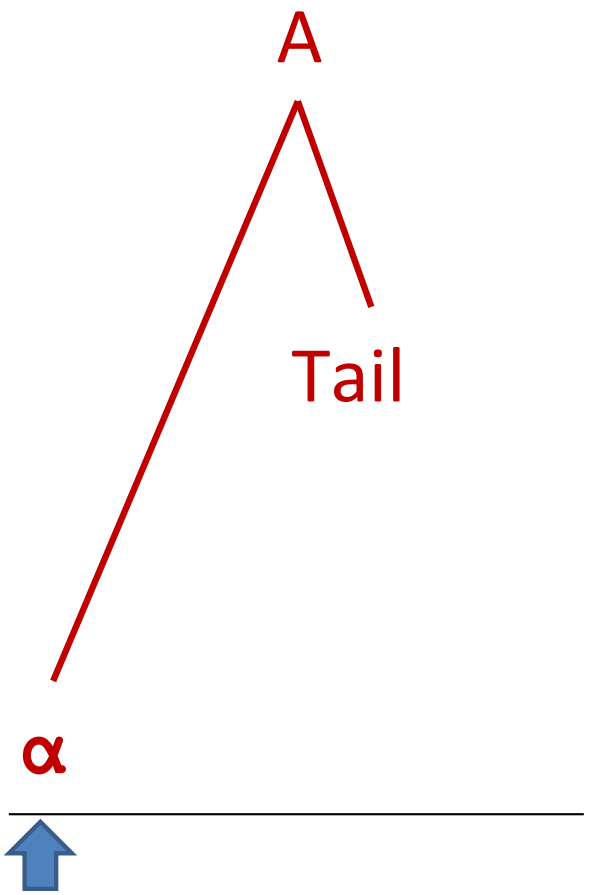
```
0. A ::= αβ | αγ
```

```
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"

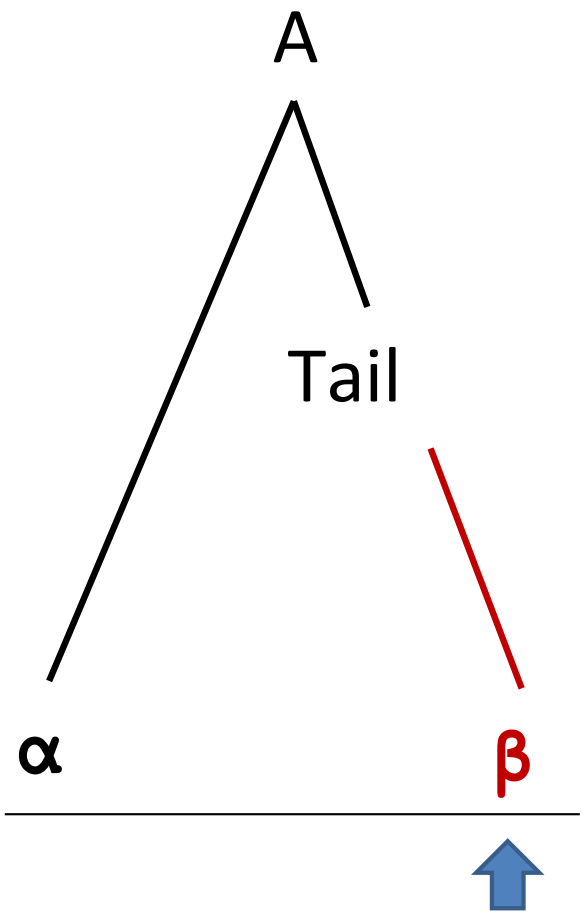



```
0. A ::= α Tail
1. Tail ::= β | γ
```

Lookahead Remaining

α β

Top-Down Derivation of "αβ"

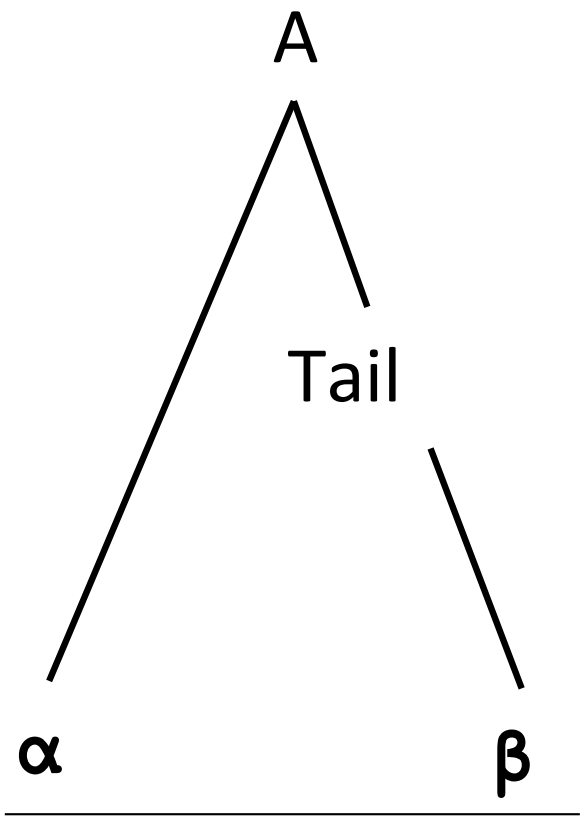



```
0. A ::= α Tail
1. Tail ::= β | γ
```

Lookahead Remaining

β

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

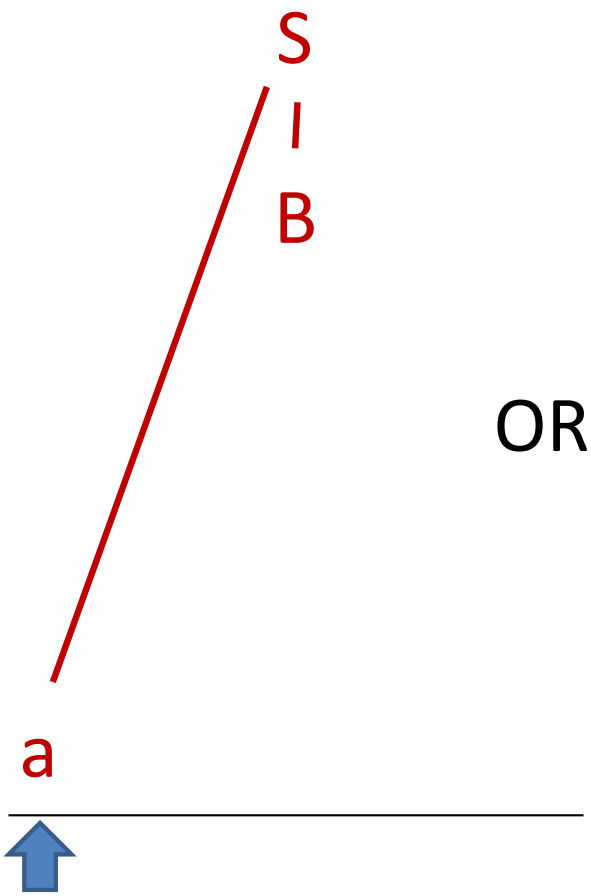
What's the issue?

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

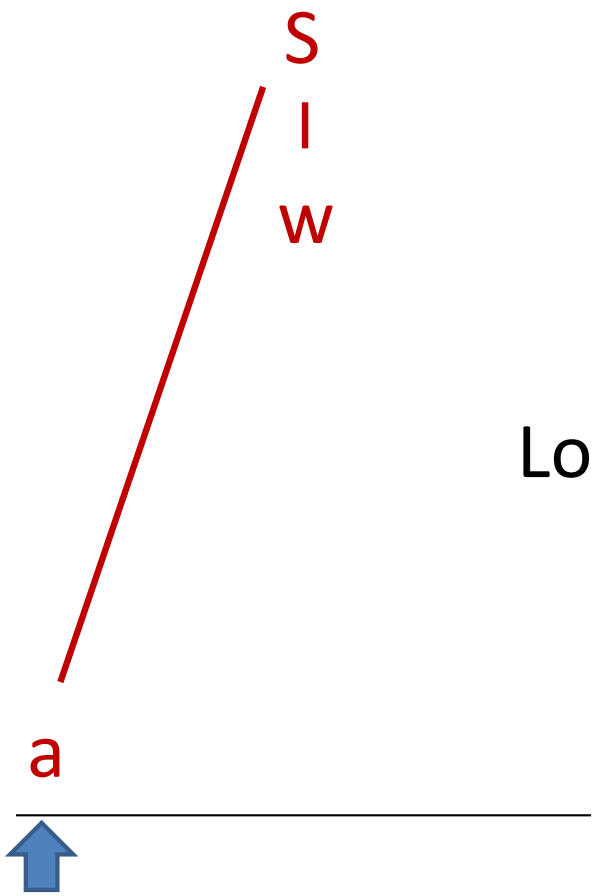
There's a FIRST Conflict!

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
```

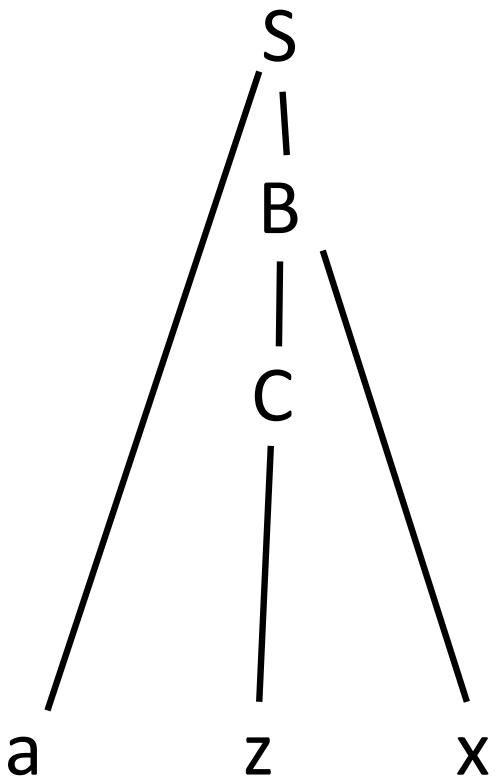


OR



Lookahead	Remaining
a	z x

Parse Tree without changing Grammar

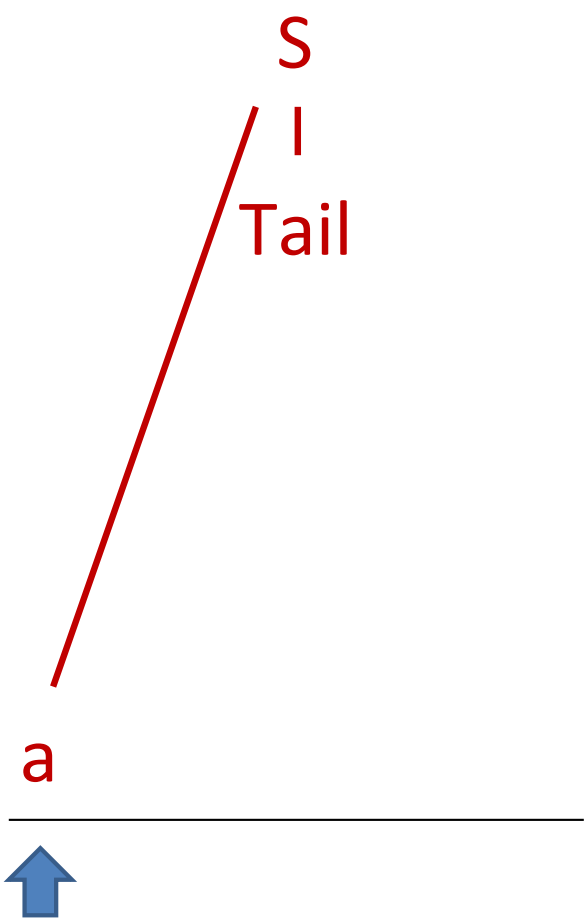



```
0. S ::= a B | a w
1. B ::= C x | y
2. C ::= ε | z
```

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

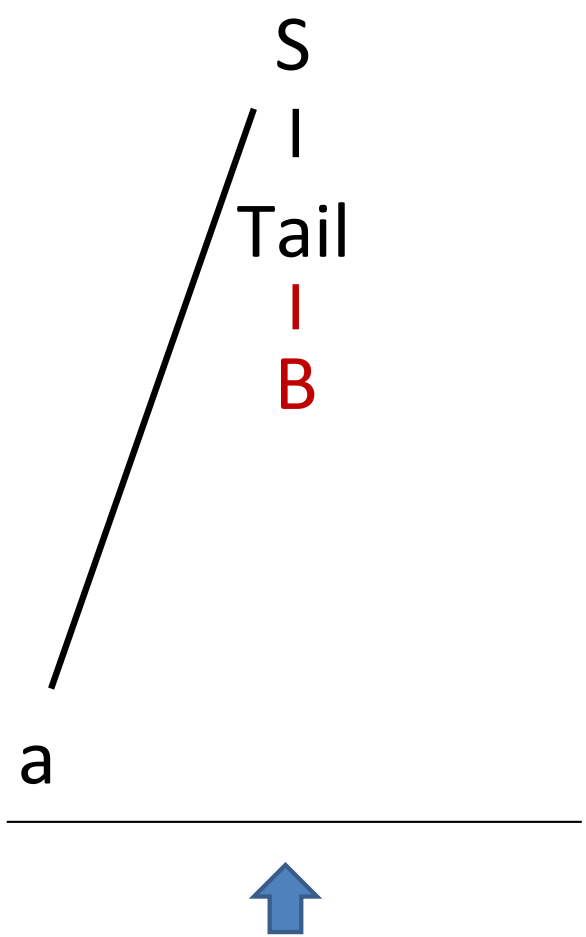
Top-Down Derivation of “a z x”




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

Lookahead	Remaining
a	z x

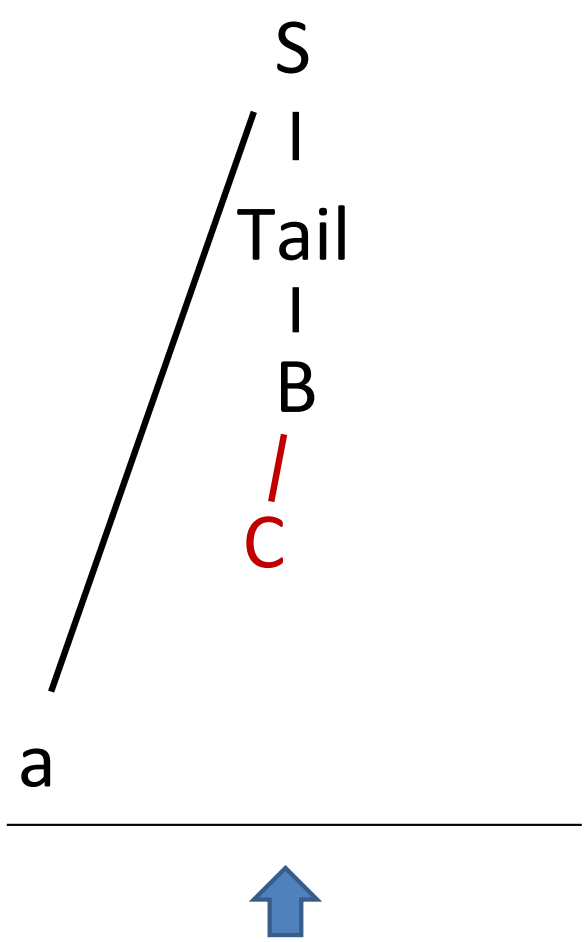
Top-Down Derivation of “a z x”




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

Lookahead	Remaining
z	x

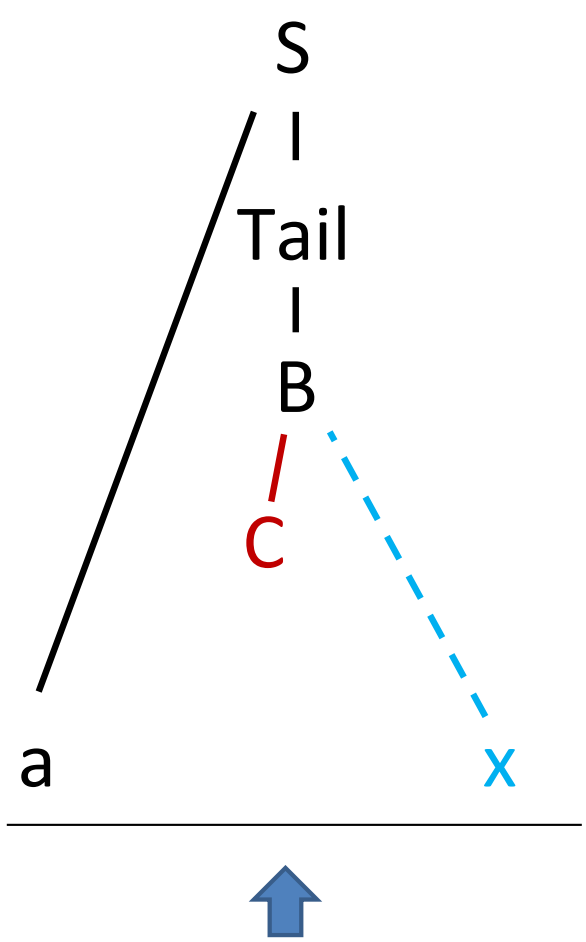
Top-Down Derivation of “a z x”




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

Lookahead	Remaining
z	x

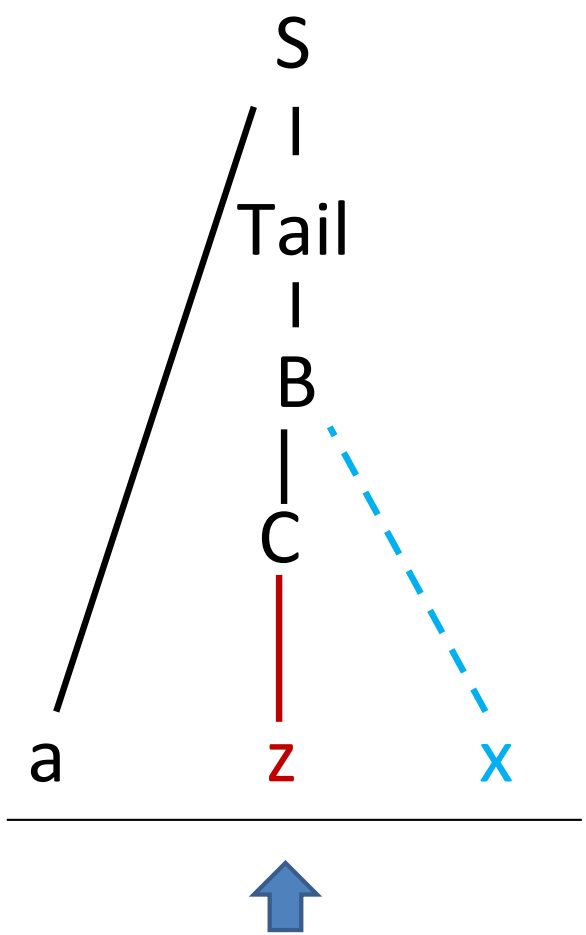
Top-Down Derivation of “a z x”




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

Lookahead	Remaining
z	x

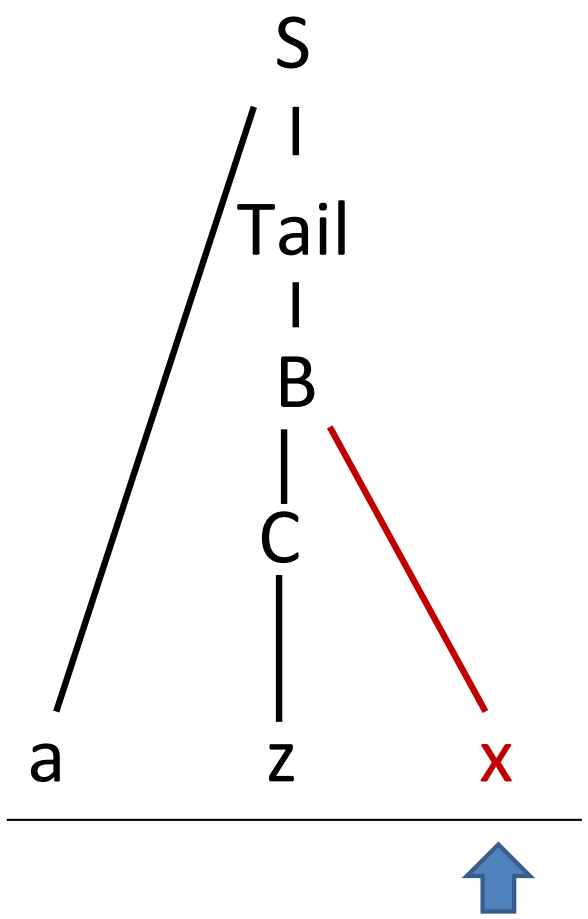
Top-Down Derivation of “a z x”




```
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”

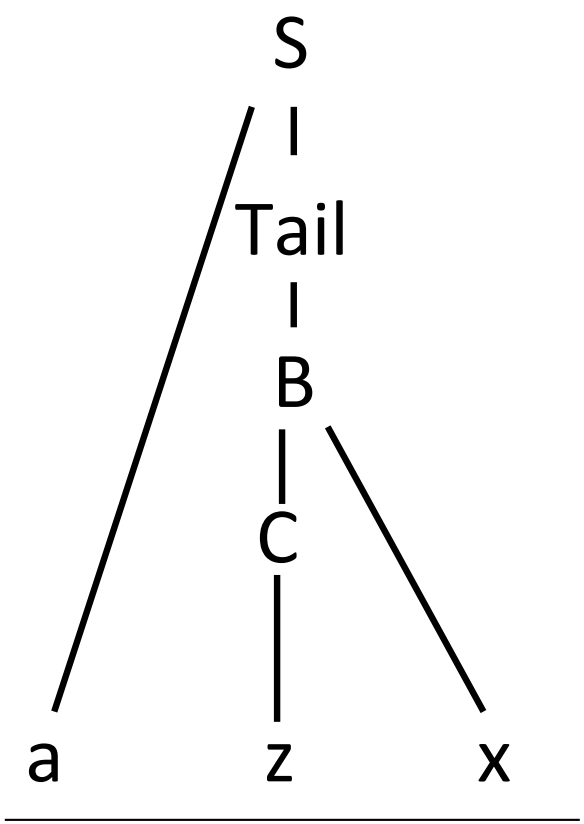


0. **S ::= a Tail**
1. **Tail ::= B | w**
2. B ::= C x | y
3. C ::= ε | z

Lookahead Remaining

x

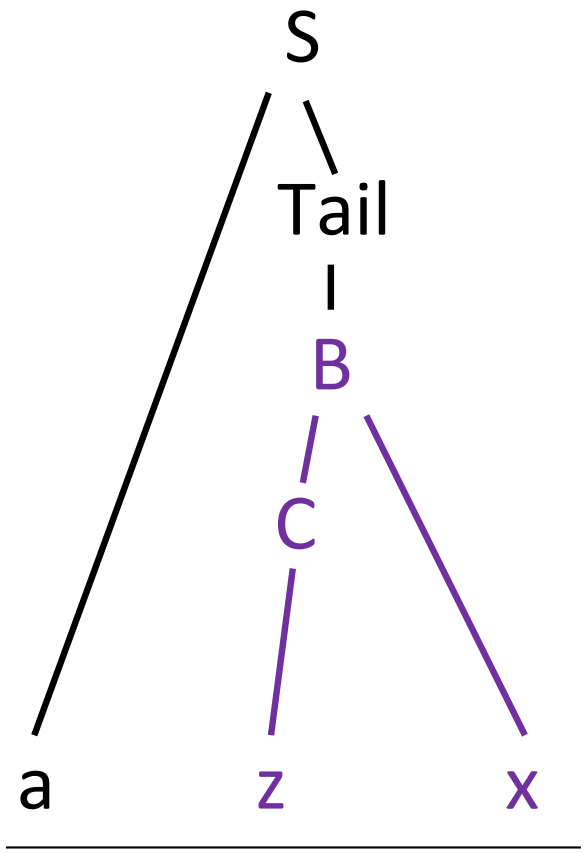
Top-Down Derivation of “a z x”



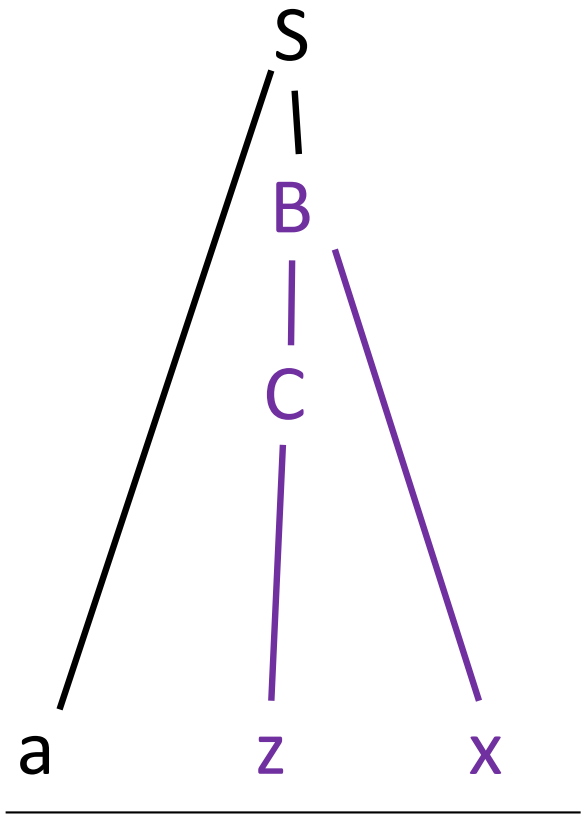
0. **S ::= a Tail**
1. **Tail ::= B | w**
2. B ::= C x | y
3. C ::= ε | z

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

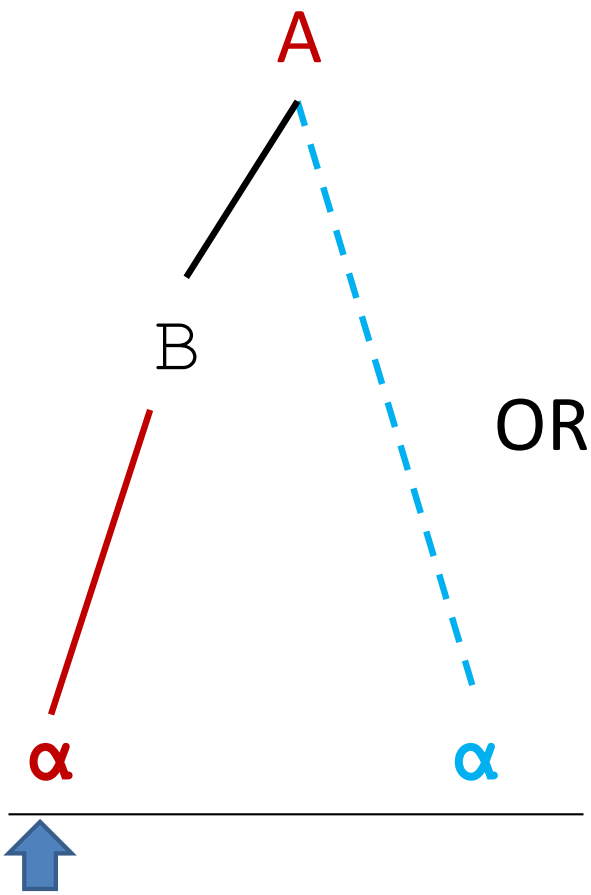
Canonical FIRST FOLLOW Conflict

Problem

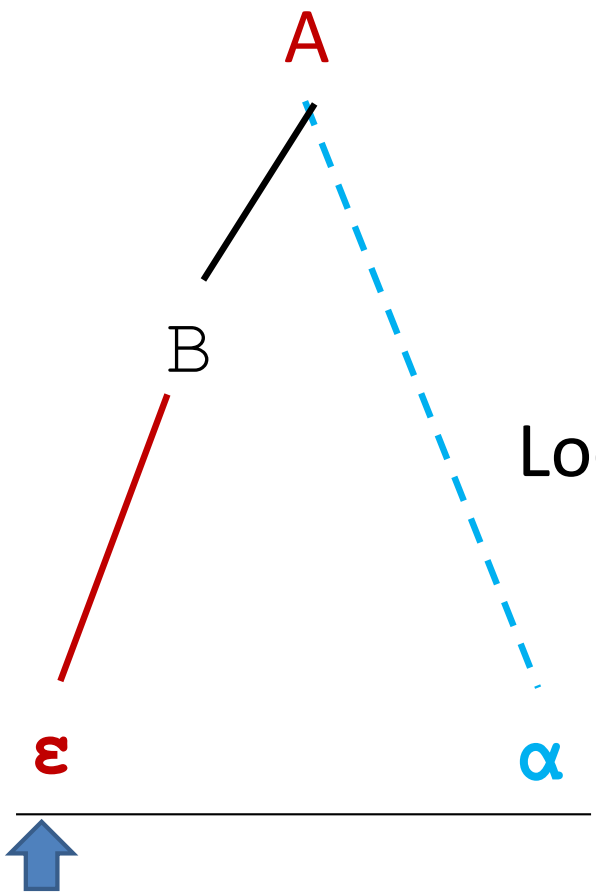
```
0. A ::= B α
1. B ::= α | ε
```

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"



OR

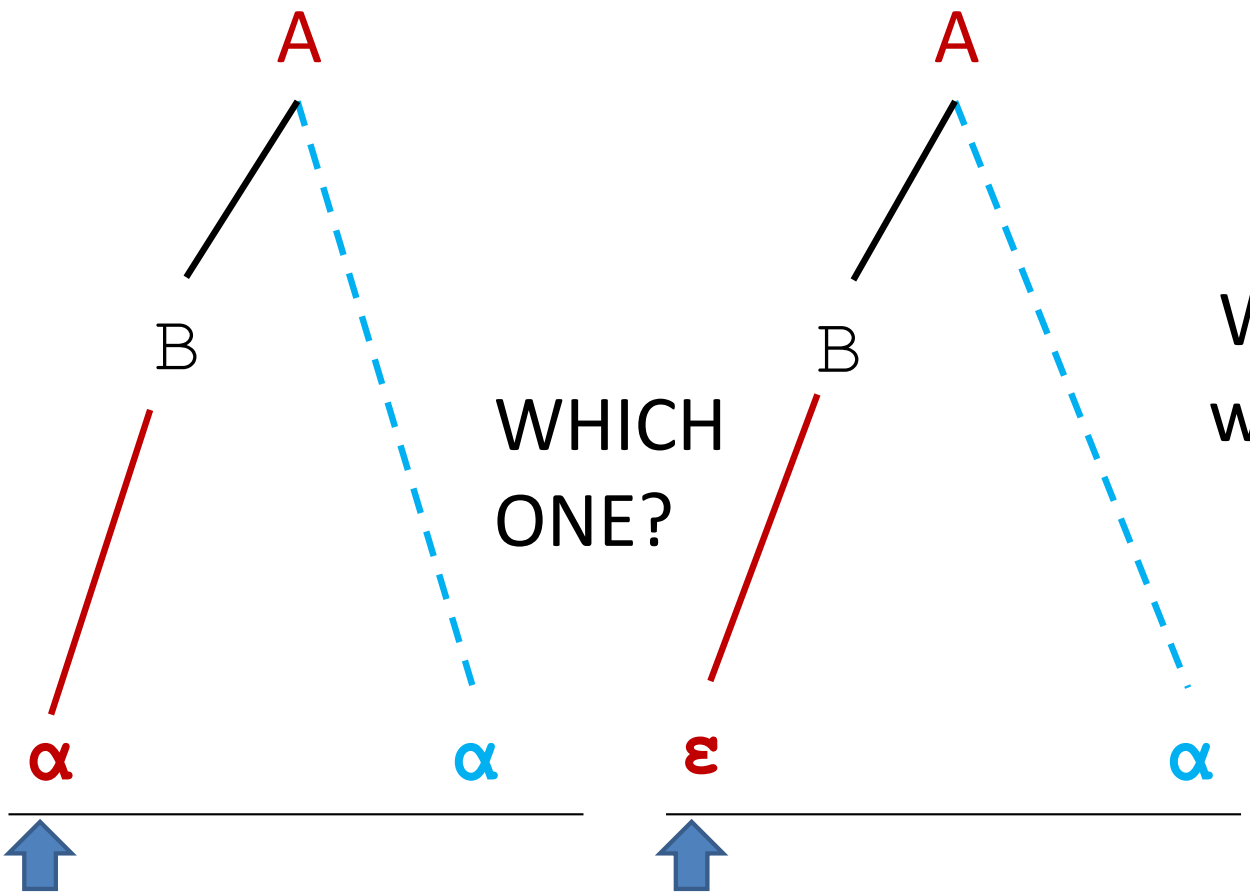



```
0. A ::= B α
1. B ::= α | ε
```

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
```

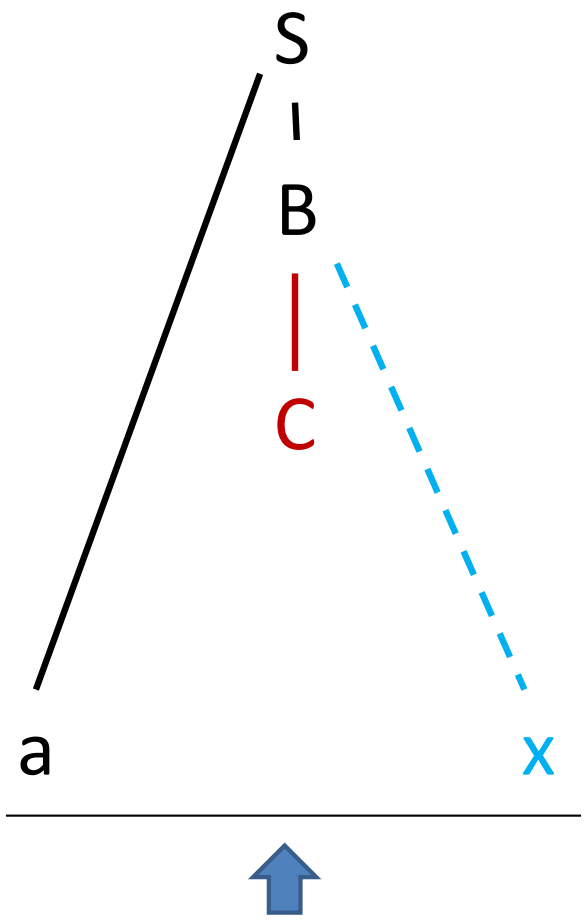
Lookahead	Remaining
a	x

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”



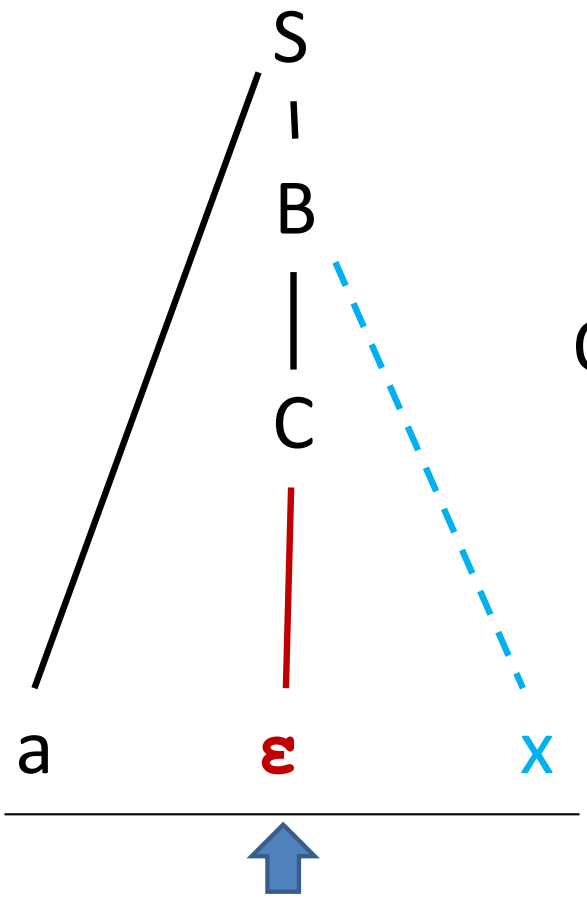

```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | x
```

Lookahead Remaining

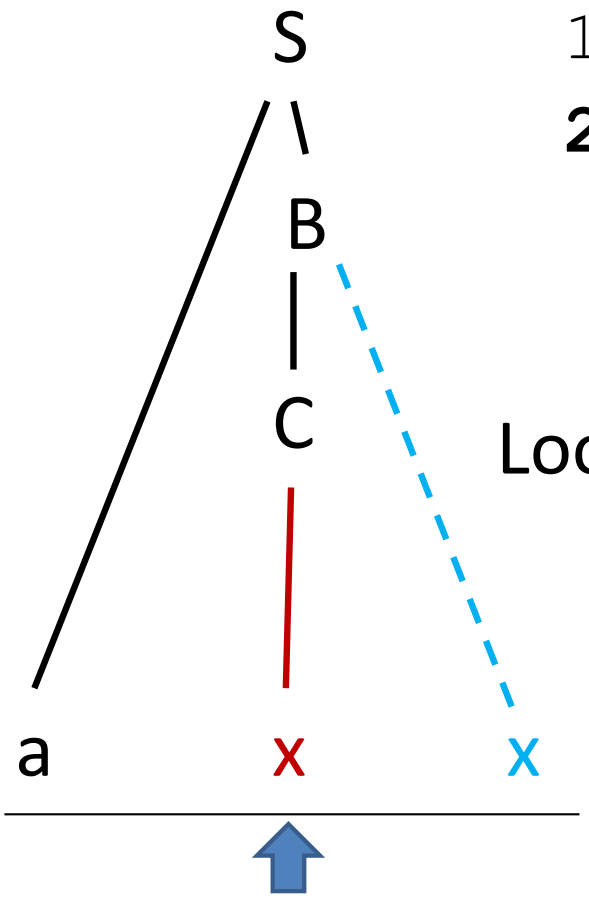
x

Top down derivation of “ax”

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



OR



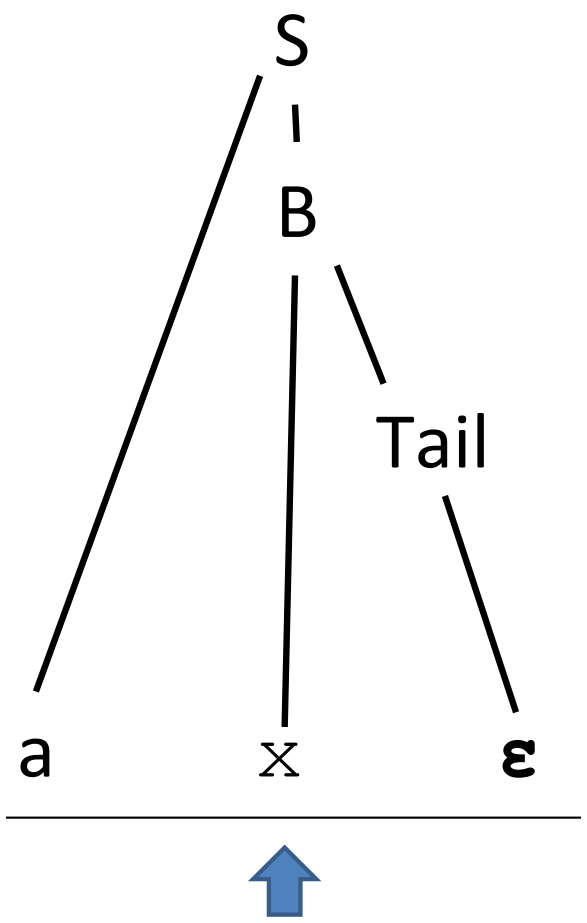
Lookahead	Remaining
x	

Applying the Fix: Substitute the Common Prefix,

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

                0. S ::= a B
2               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
```

Lookahead Remaining

x