More on LL Grammars + Interpreters

Kris, Michael, Gavin, Anand, Eunia

CSE 401/M501 – Section 5

Autumn 2020

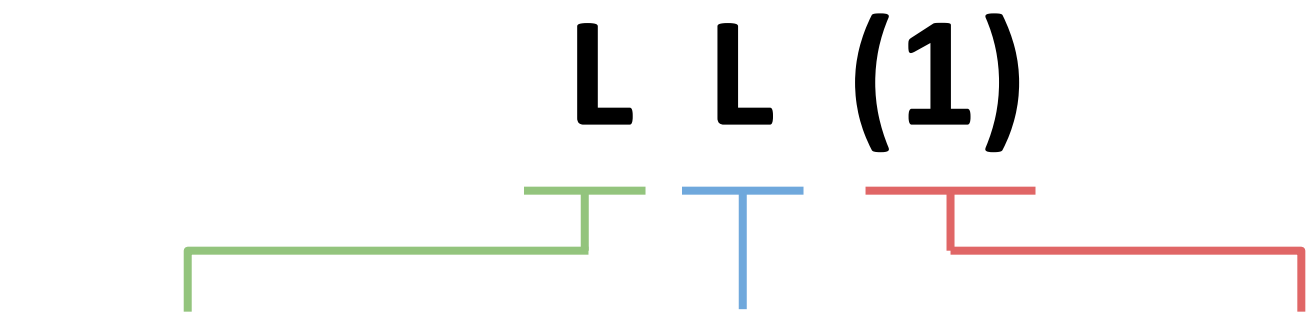
Announcements

- Parser & AST due ***tonight***!

- Homework 3 (LL grammars) due next Thursday
  - *Please start early – the programming portion may take a while!*

Agenda

- **What LL (Top-Down) Parsing Looks Like**
- LL Grammar Issues
  - FIRST Conflict
  - FIRST FOLLOW Conflict
  - Recursion
  - Indirect Left Recursion
- Interpreters vs Compilers



Left-to-Right

Only takes one pass, performed from the left

Leftmost

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

1 Terminal Lookahead

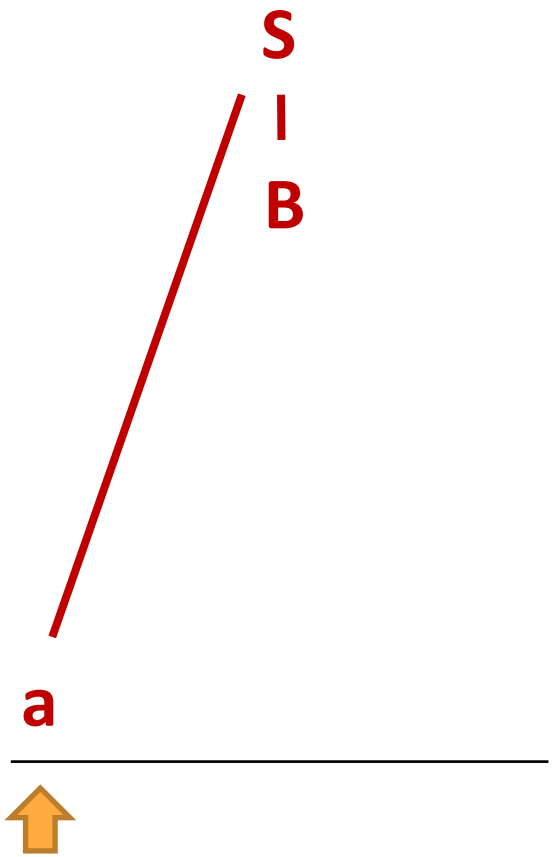
Must determine derivation from the next unparsed terminal in the string

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**

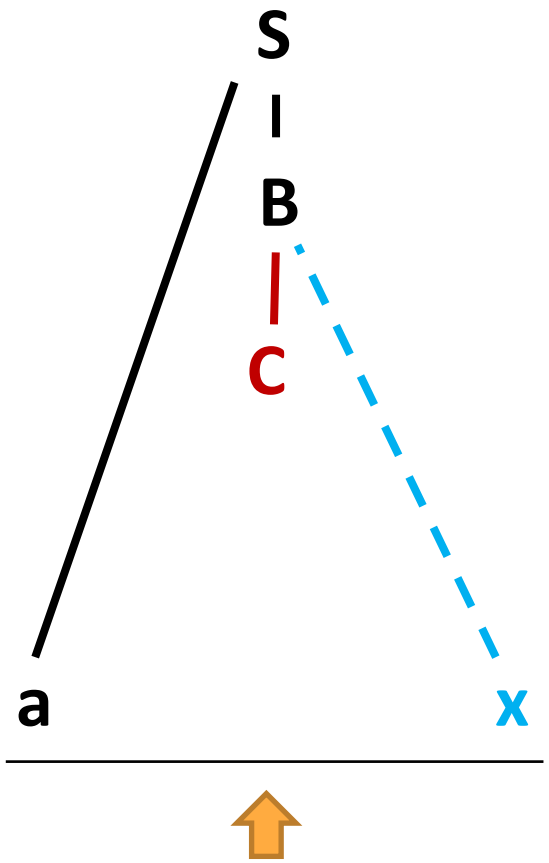
Top-Down Derivation of “a z x”




```
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”

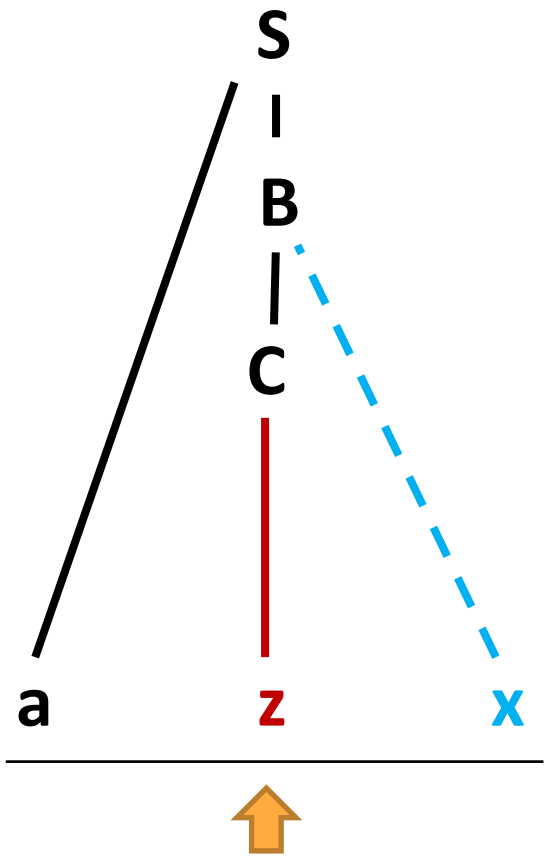



```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | z
```

Lookahead Remaining

z x

Top-Down Derivation of “a z x”

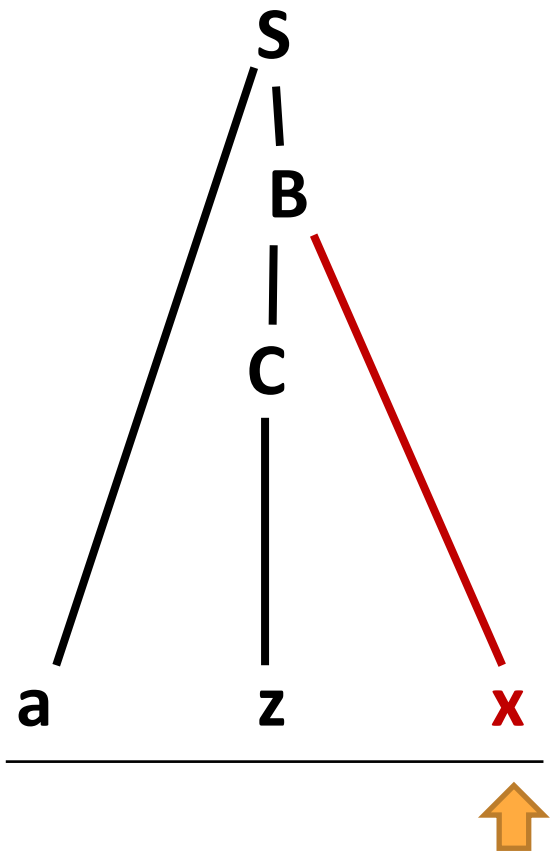



```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | z
```

Lookahead Remaining

z x

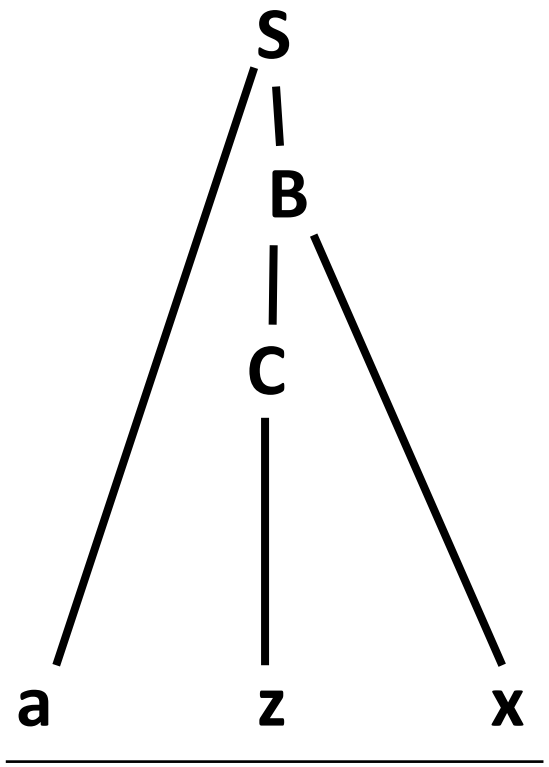
Top-Down Derivation of “a z x”




```
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!

Agenda

- What LL (Top-Down) Parsing Looks Like
- LL Grammar Issues
  - **FIRST Conflict**
  - FIRST FOLLOW Conflict
  - Recursion
  - Indirect Left Recursion
- Interpreters vs Compilers

Canonical FIRST Conflict

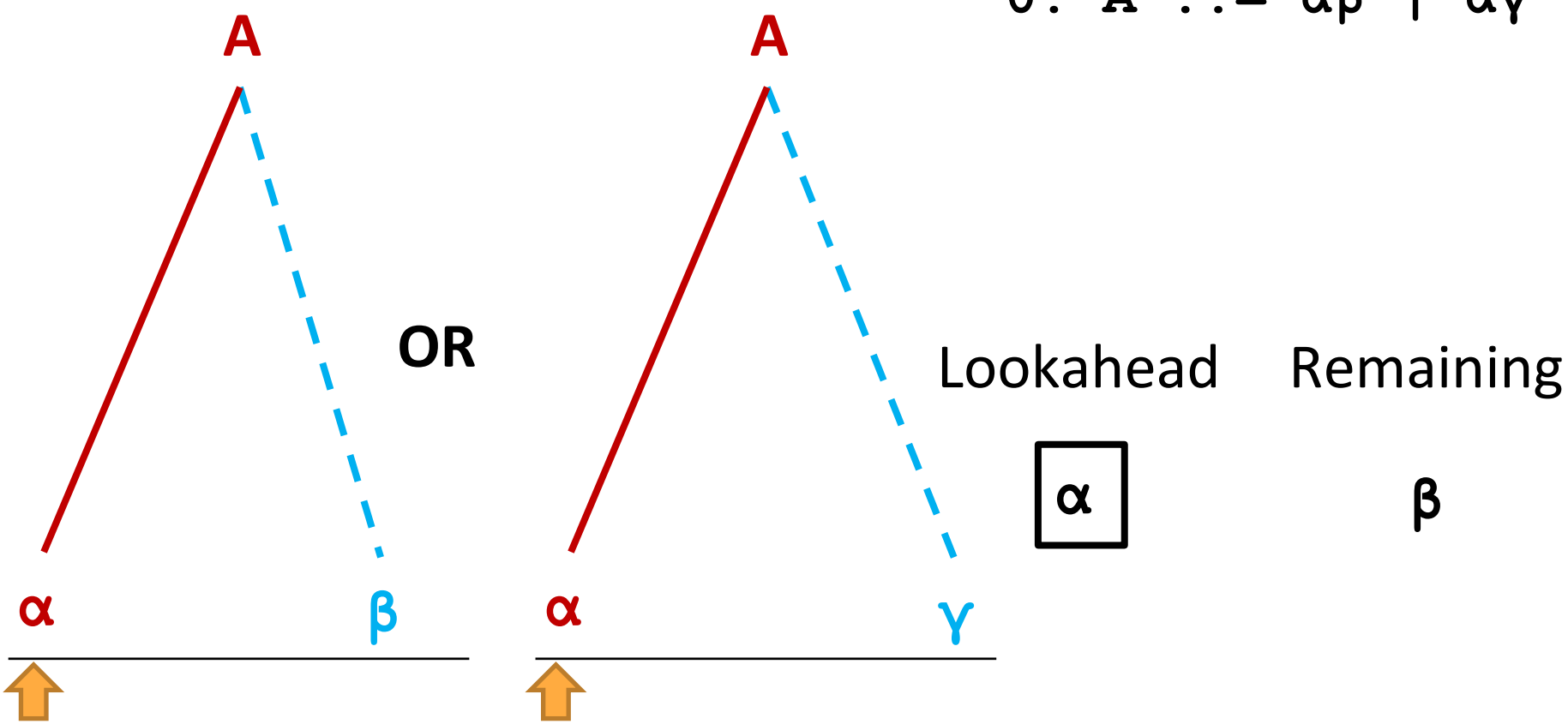
Problem

0. **A** ::= αβ | αγ

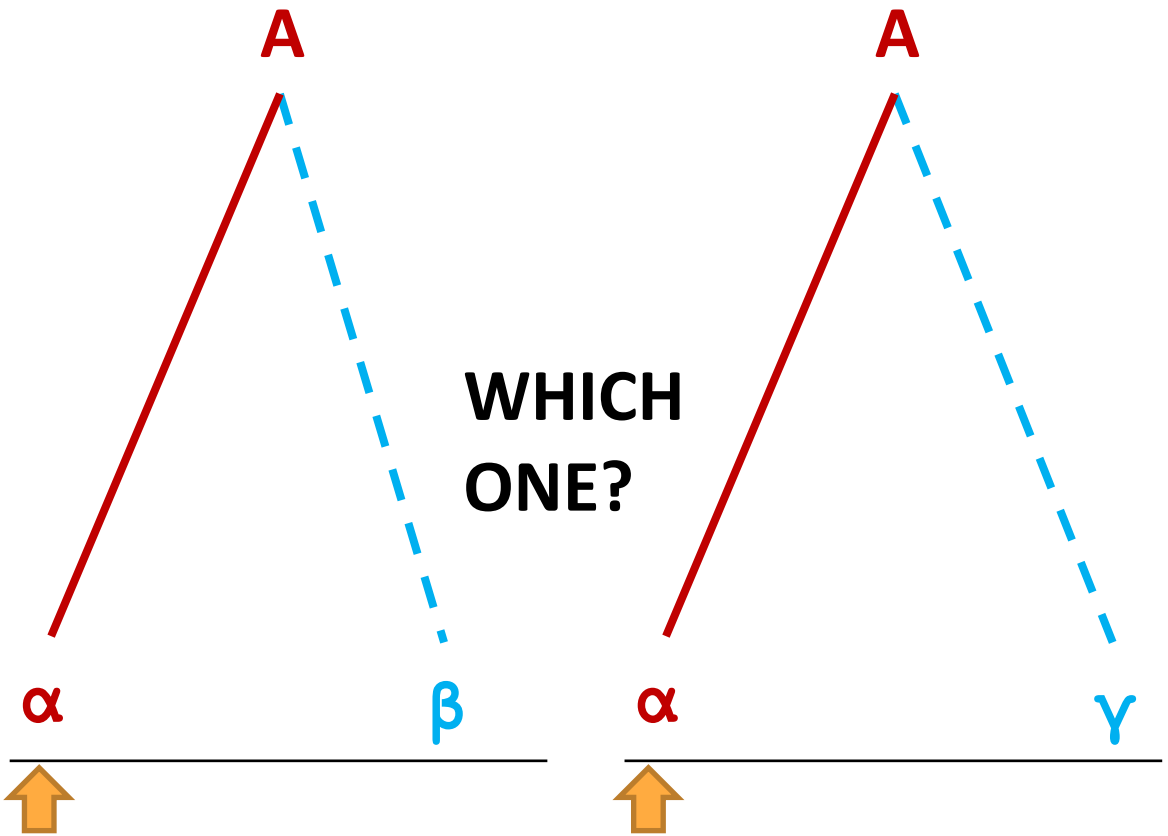
The FIRST sets of the right-hand sides for the **SAME NON-TERMINAL** must be disjoint!

Let's try a top-down derivation of αβ

```
0. A ::= αβ | αγ
```



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

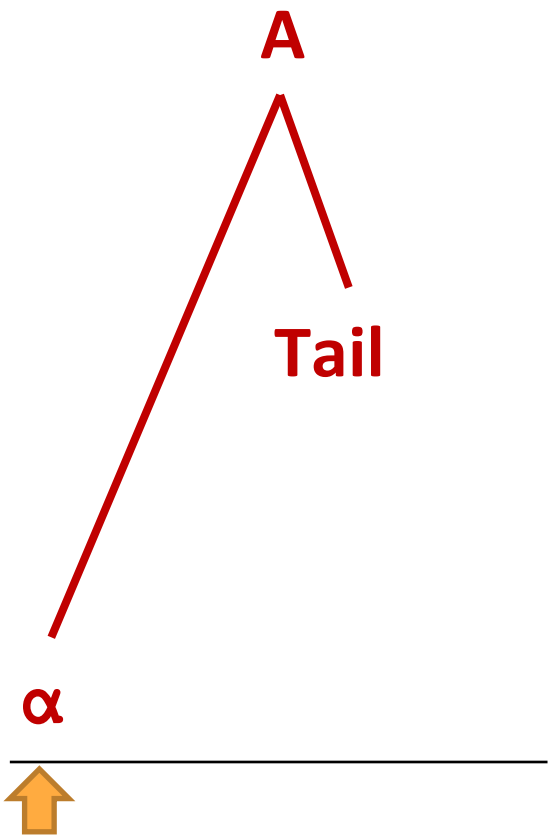
Solution

0. **A** ::= **αβ** | **αγ**

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

Factor out the common prefix

Top-Down Derivation of "αβ"

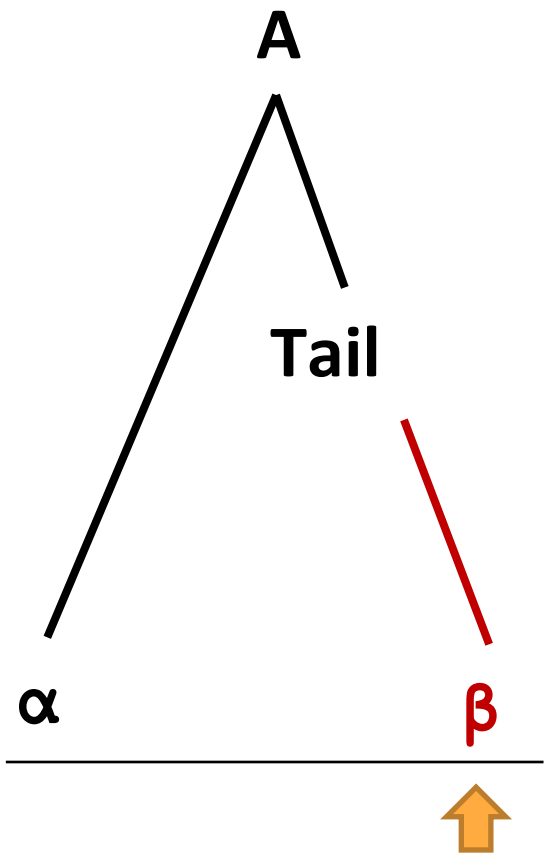



```
0. A ::= α Tail
1. Tail ::= β | γ
```

Lookahead Remaining

α β

Top-Down Derivation of "αβ"

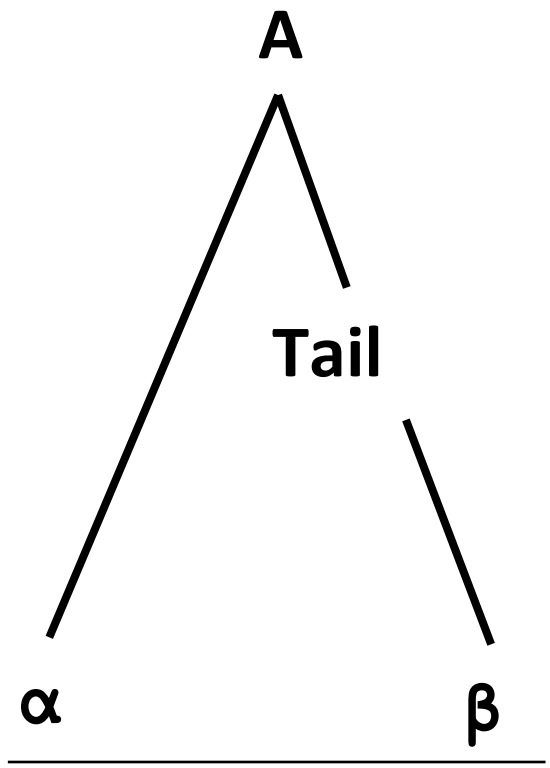



```
0. A ::= α Tail
1. Tail ::= β | γ
```

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

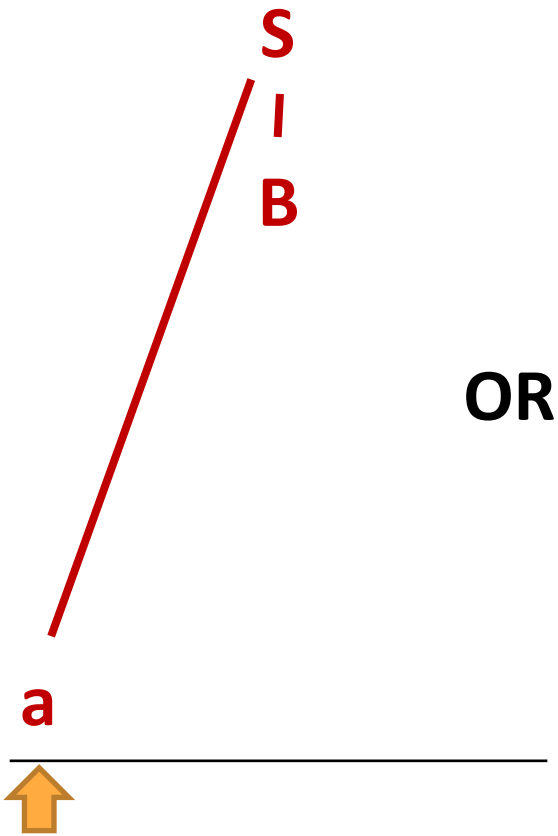
What's the issue?

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

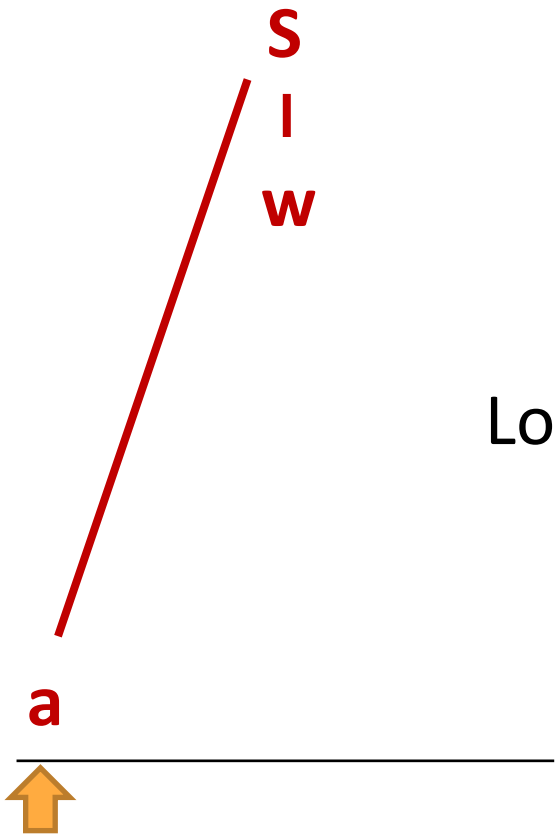
There's a FIRST Conflict!

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
```

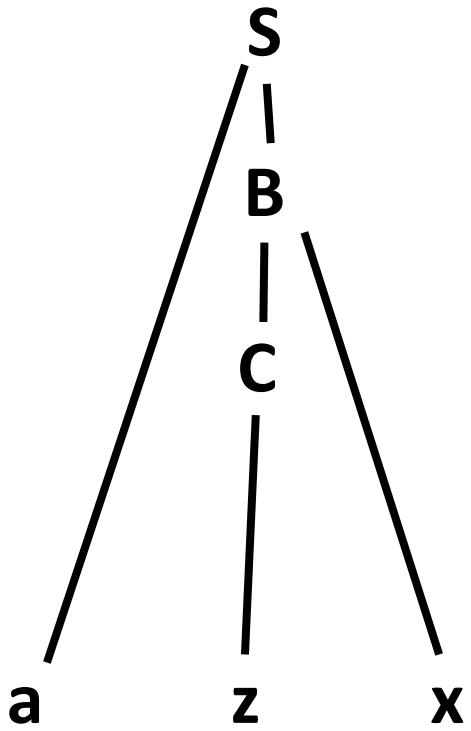


OR



Lookahead	Remaining
a	z x

Parse Tree without changing Grammar

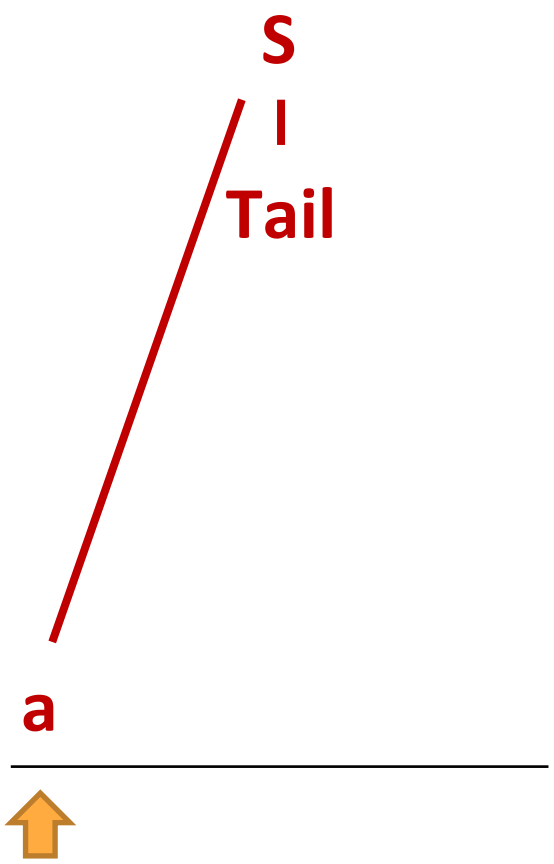



```
0. S ::= a B | a w
1. B ::= C x | y
2. C ::= ε | z
```

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

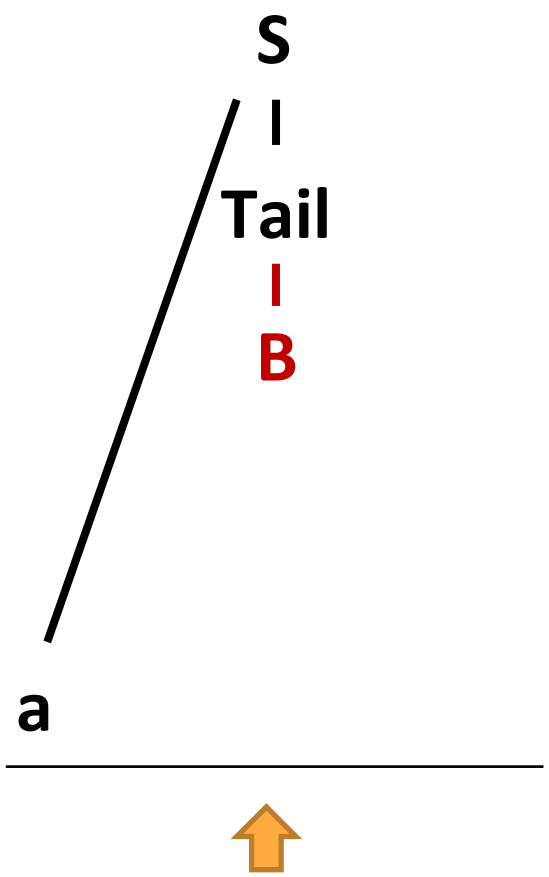
Top-Down Derivation of “a z x”




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

Lookahead	Remaining
a	z x

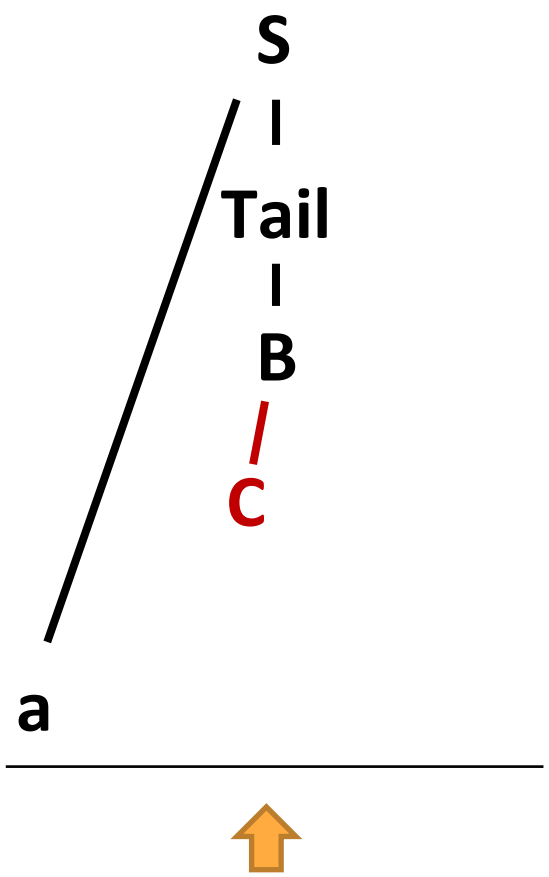
Top-Down Derivation of “a z x”




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

Lookahead	Remaining
z	x

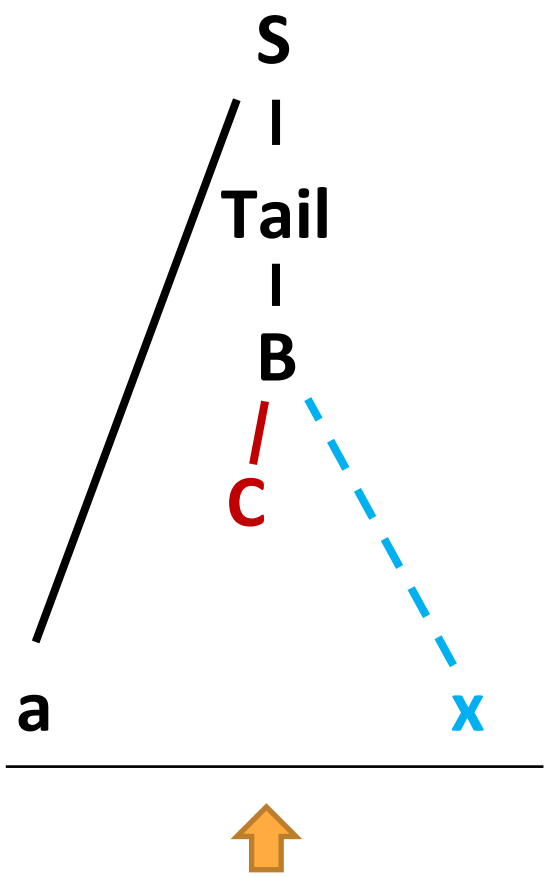
Top-Down Derivation of “a z x”




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

Lookahead	Remaining
z	x

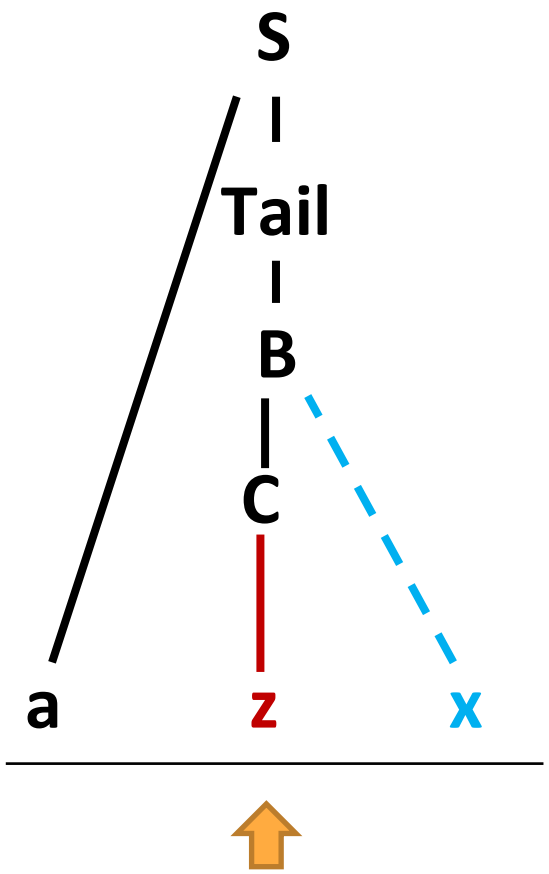
Top-Down Derivation of “a z x”




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

Lookahead	Remaining
z	x

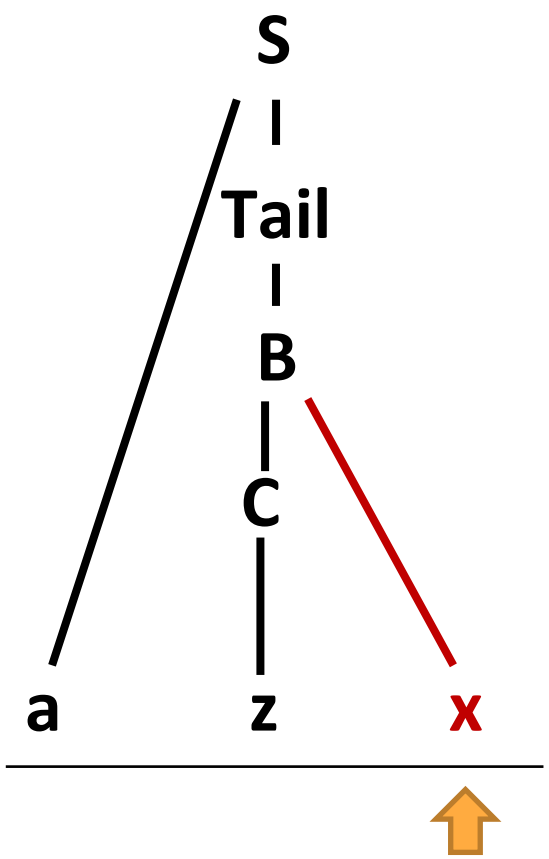
Top-Down Derivation of “a z x”




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

Lookahead	Remaining
z	x

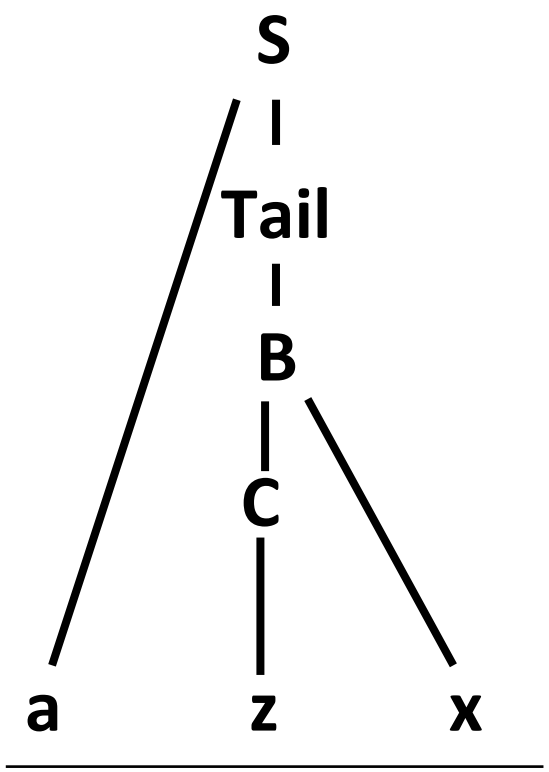
Top-Down Derivation of “a z x”




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

Lookahead Remaining

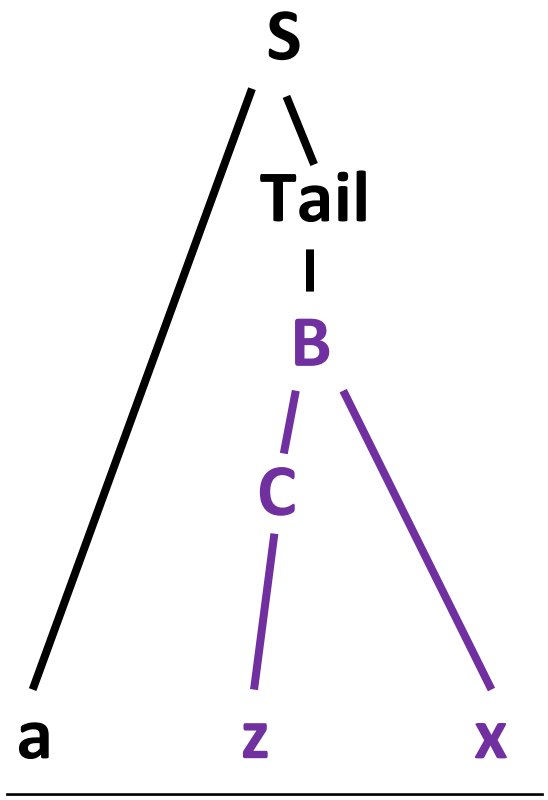
Top-Down Derivation of “a z x”



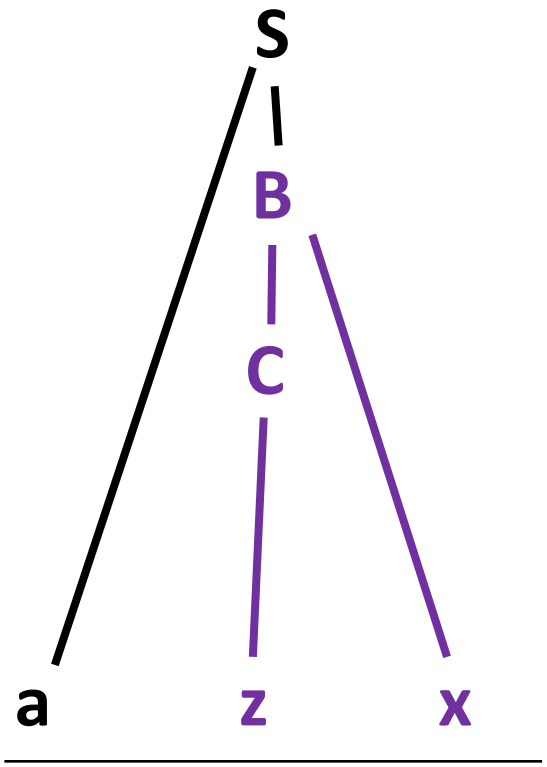
0. **S ::= a Tail**
1. **Tail ::= B | w**
2. B ::= C x | y
3. C ::= ε | z

Success!

Comparing Parse Trees



Purple trees are the same!



Agenda

- What LL (Top-Down) Parsing Looks Like
- LL Grammar Issues
  - FIRST Conflict
  - **FIRST FOLLOW Conflict**
  - Left Recursion
  - Indirect Left Recursion
- Interpreters vs Compilers

Canonical FIRST FOLLOW Conflict

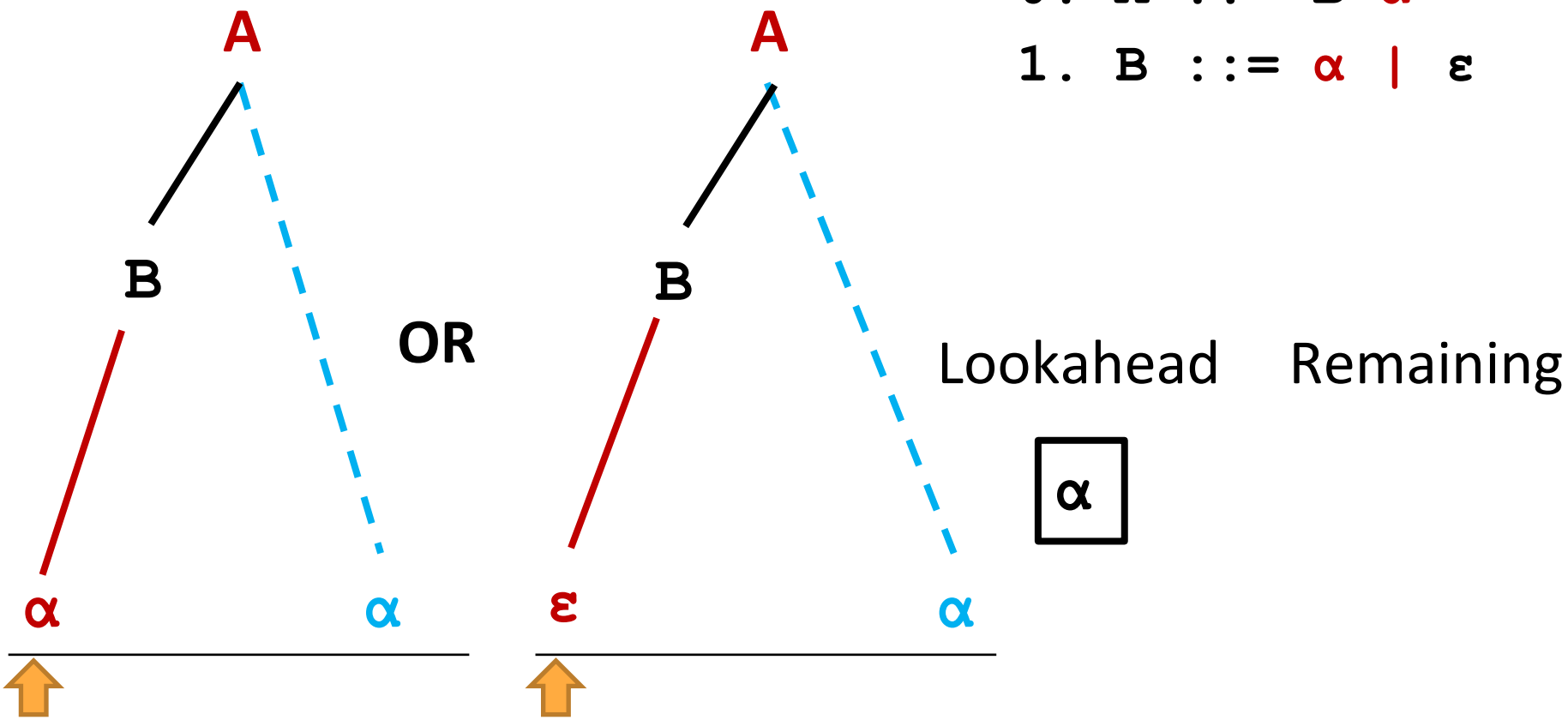
Problem

```
0. A ::= B α
1. B ::= α | ε
```

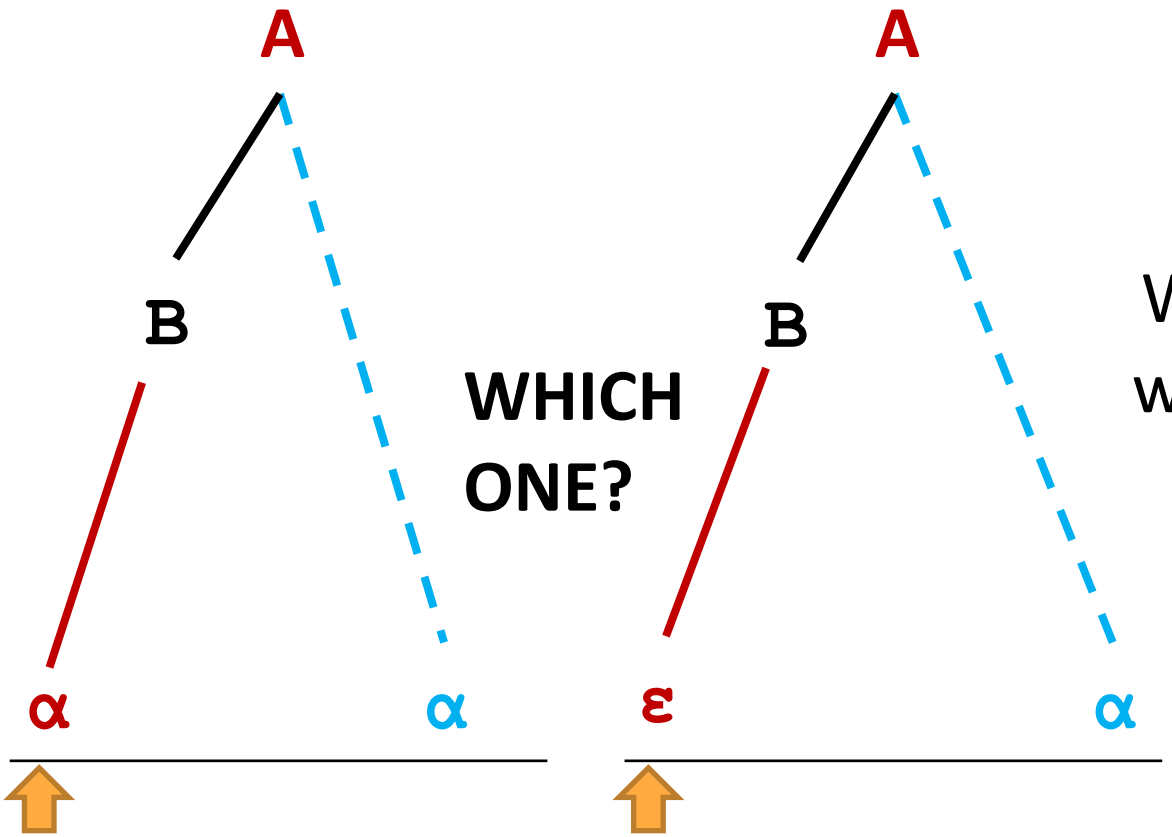
Because B is nullable, its FOLLOW set must be disjoint from the FIRST sets of its right-hand sides!

Let's try a top-down derivation of "α"

```
0. A ::= B α
1. B ::= α | ε
```



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
```

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

Factor out the tail

Watch out for Nullability! (Grammar 2)

Changing the grammar again…

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

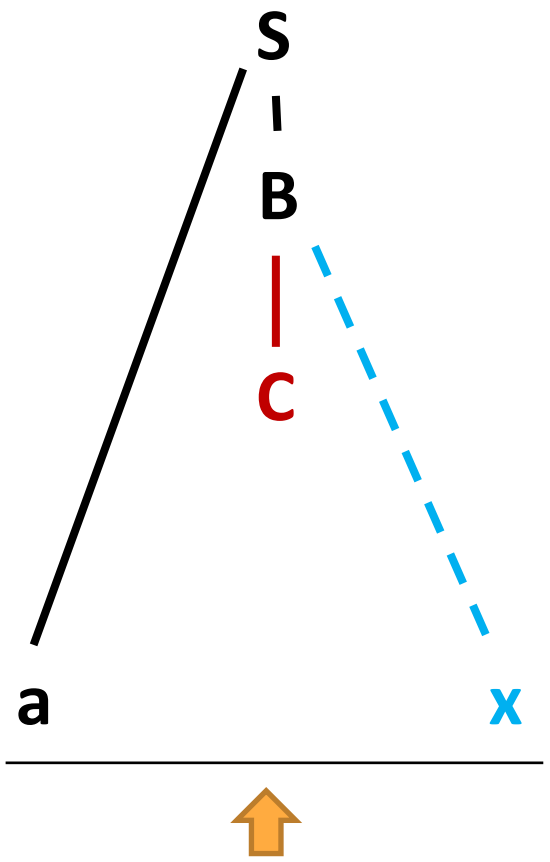
Lookahead	Remaining
a	x

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”



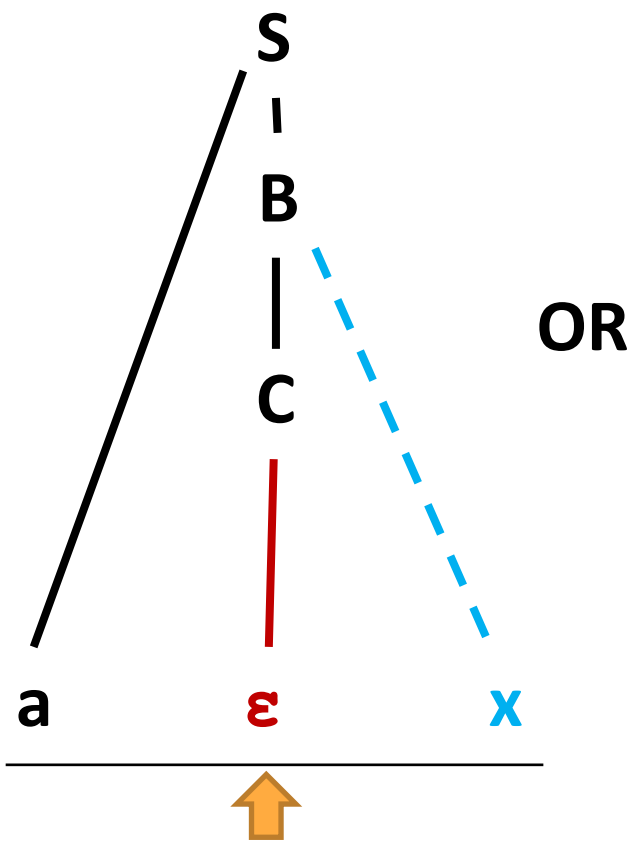

```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | x
```

Lookahead Remaining

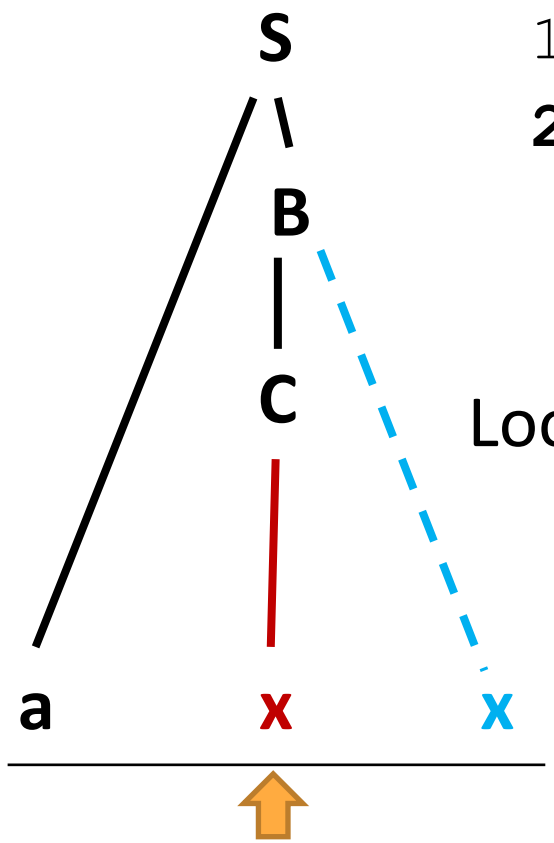
x

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, then Factor

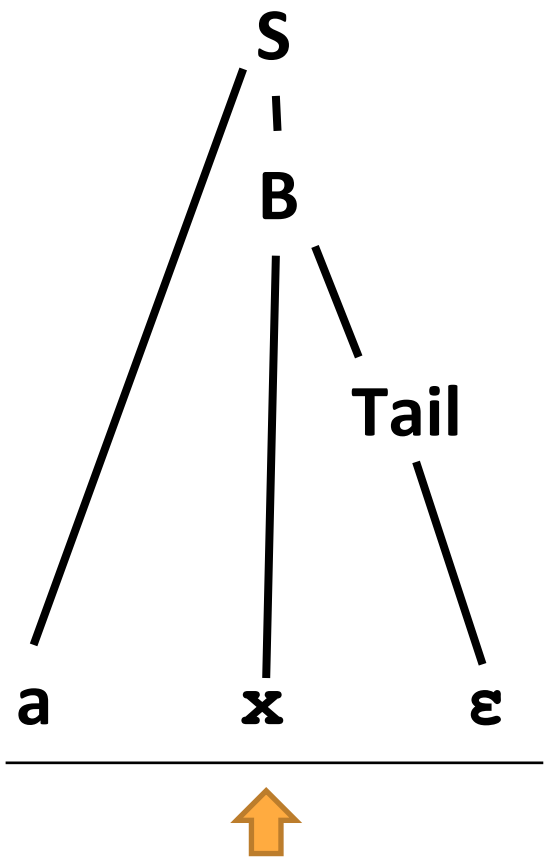
1

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
```

Lookahead Remaining

Agenda

- What LL (Top-Down) Parsing Looks Like
- LL Grammar Issues
  - FIRST Conflict
  - FIRST FOLLOW Conflict
  - **Left Recursion**
  - Indirect Left Recursion
- Interpreters vs Compilers

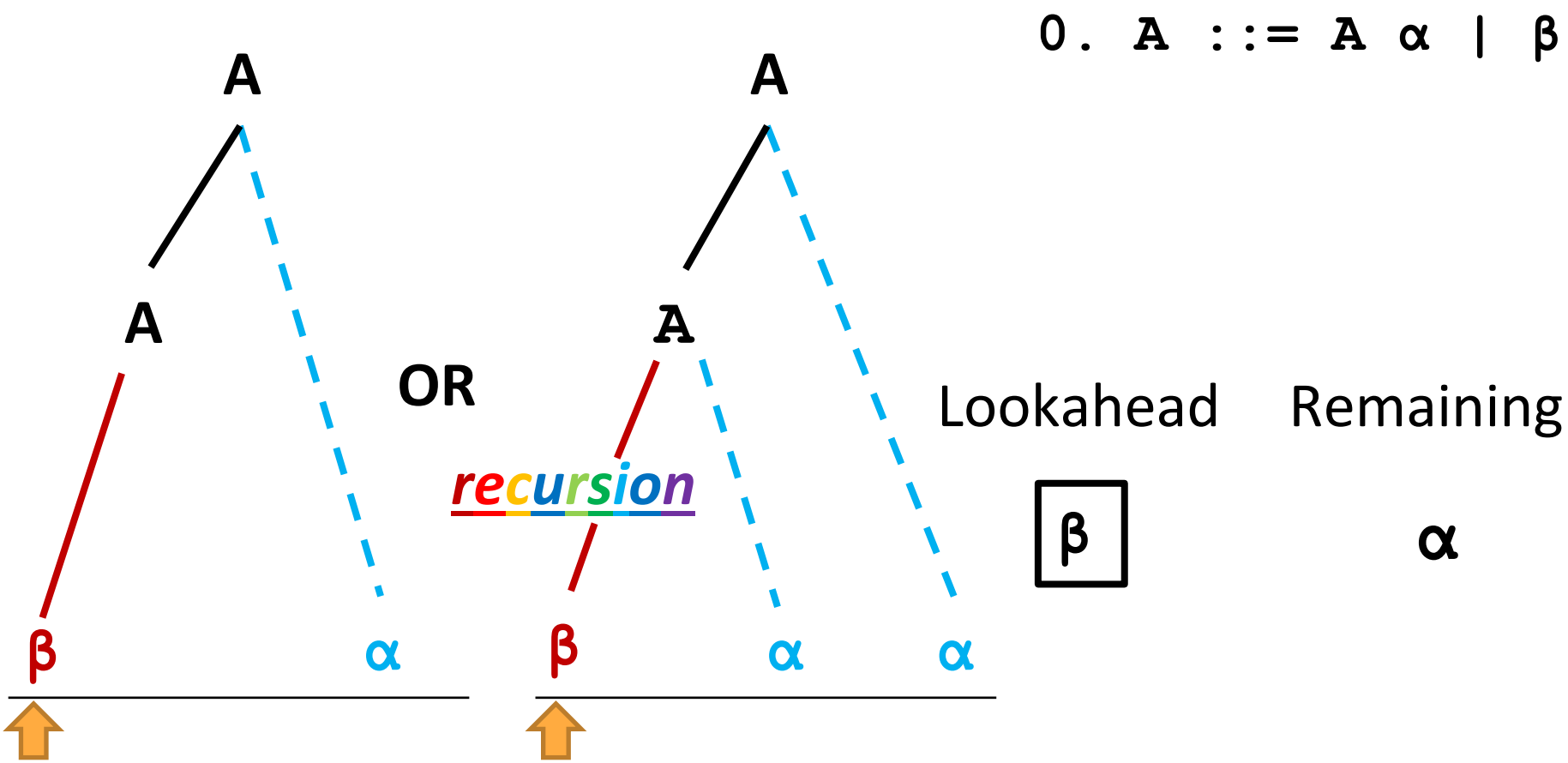
Canonical Left Recursion Problem

Problem

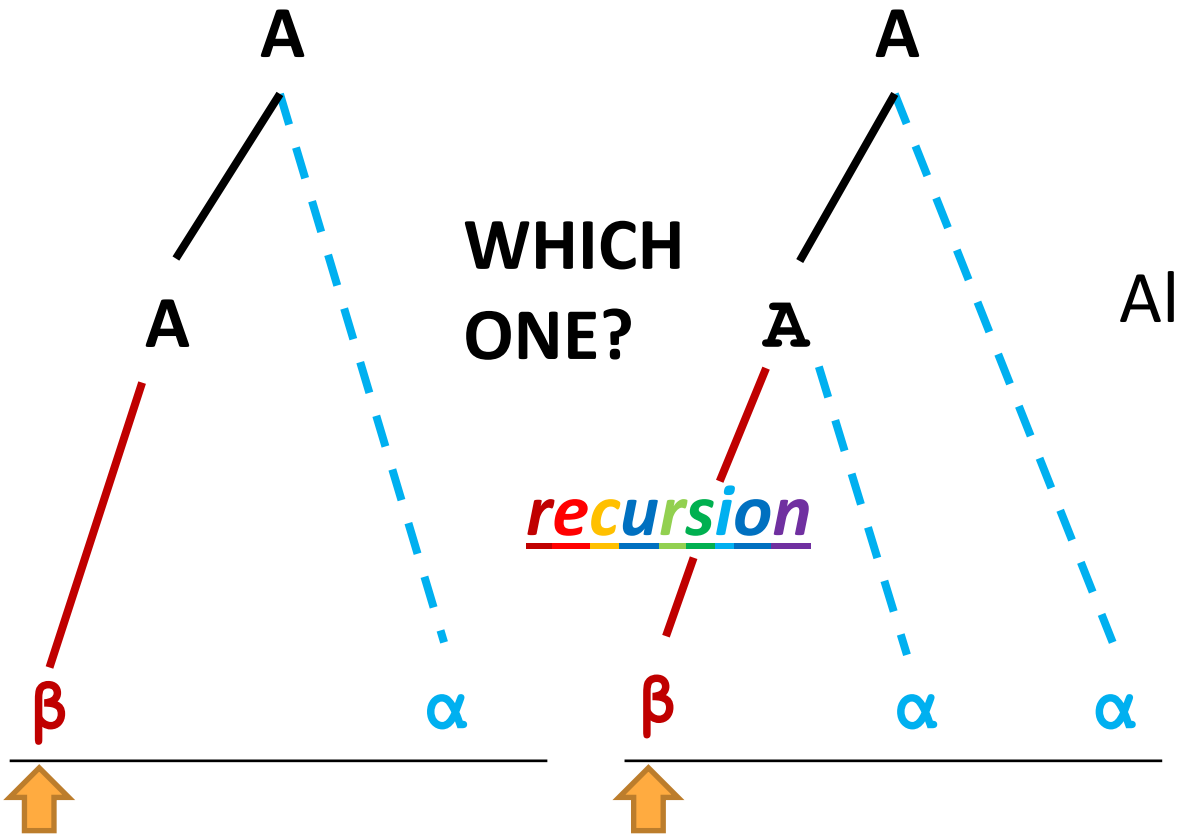
```
0. A ::= A α | β
```

Left recursion can’t be parsed by LL(1) parsers!

Let's try a top-down derivation of "βα"



Let's try a top-down derivation of "βα"




```
0. A ::= A α | β
```

All we know right now is that it starts with **β!**

Canonical Left Recursion Solution

Solution

```
0. A ::= A α | β
```

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

1. Turn suffix of all A's recursive rhs into a tail non-terminal.
2. Append the tail to every non-recursive rhs.
3. Add the empty string (ε) as a rhs for the tail production.

Let's change the grammar again! (Grammar 3)

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

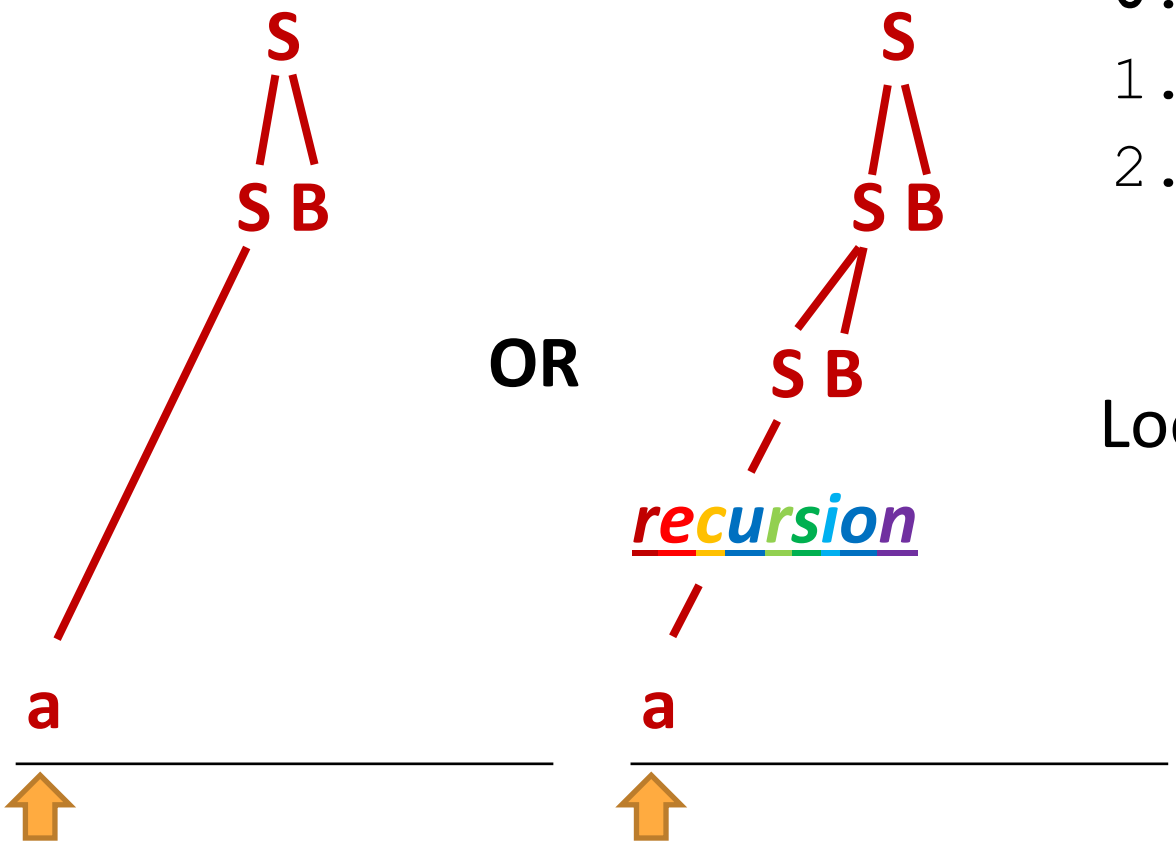
Lookahead	Remaining
a	z x

What's the issue?

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

Left recursion can't be parsed by LL(1) parsers!

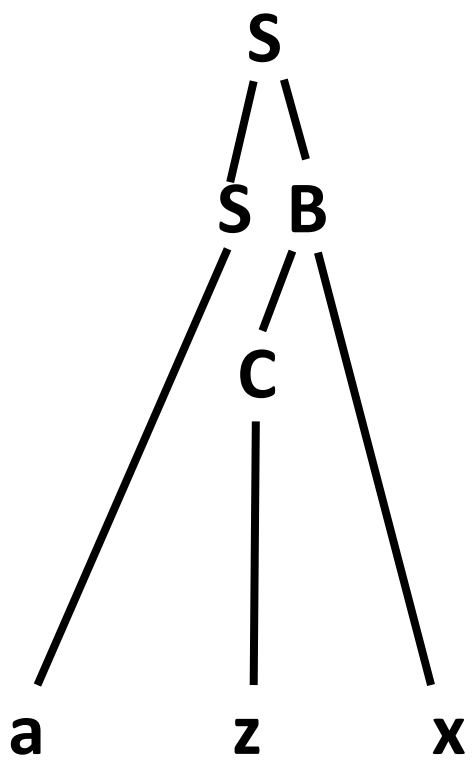
Top-Down Derivation of “a z x”




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

Lookahead	Remaining
a	z x

Parse Tree without changing Grammar

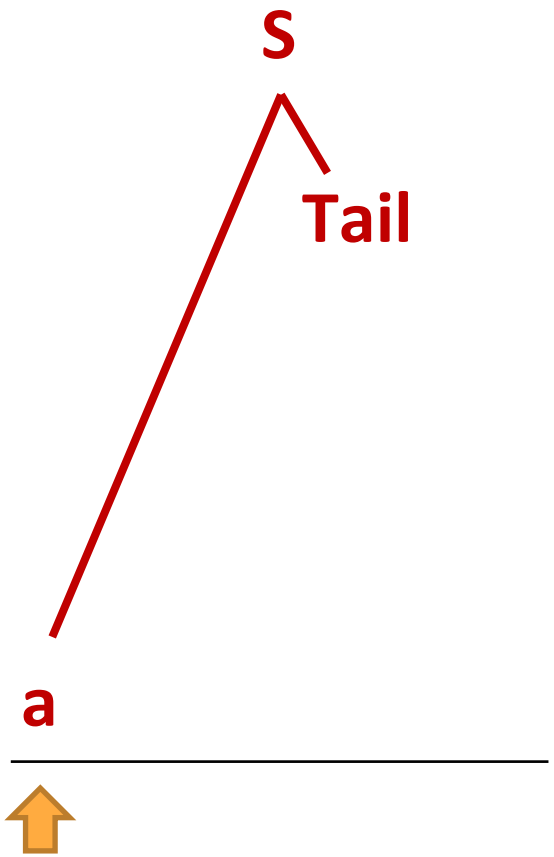



```
0. S ::= S B | a | w
1. B ::= C x | y
2. C ::= ε | z
```

Applying the fix:

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

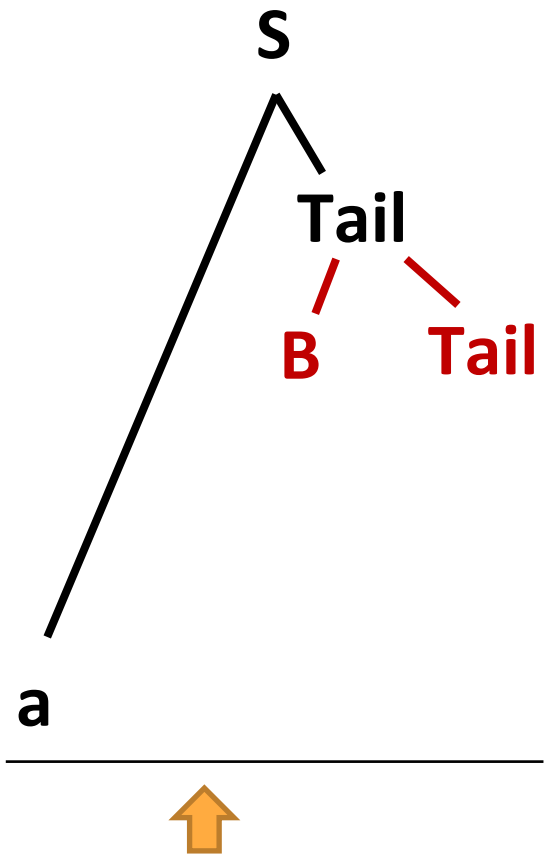
Top-Down Derivation of “a z x”




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

Lookahead	Remaining
a	z x

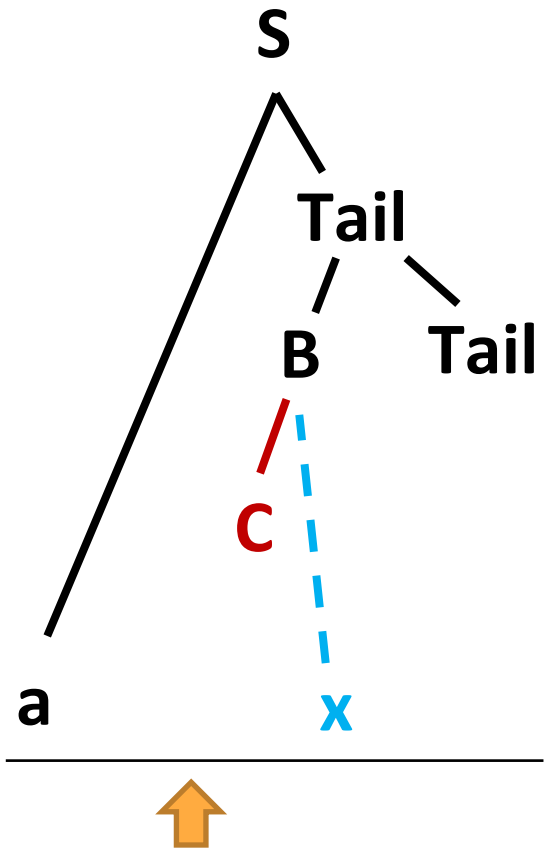
Top-Down Derivation of “a z x”




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

Lookahead	Remaining
z	x

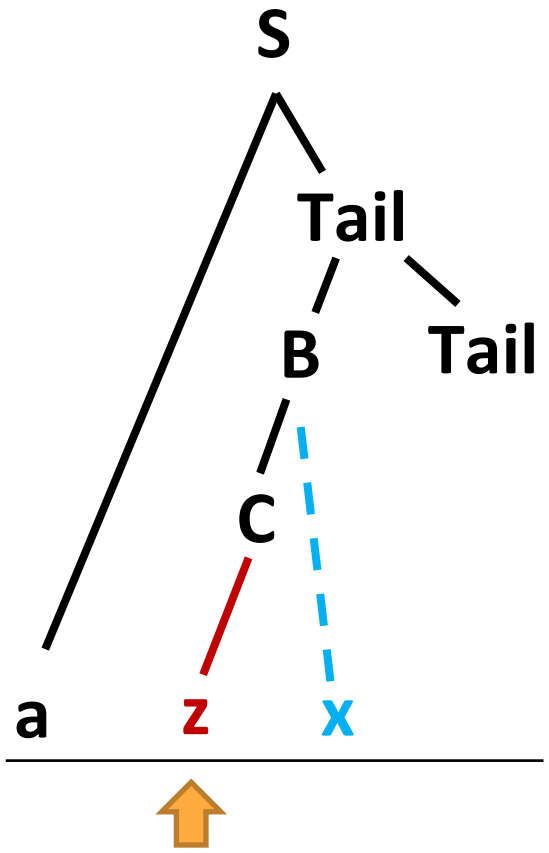
Top-Down Derivation of “a z x”




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

Lookahead	Remaining
z	x

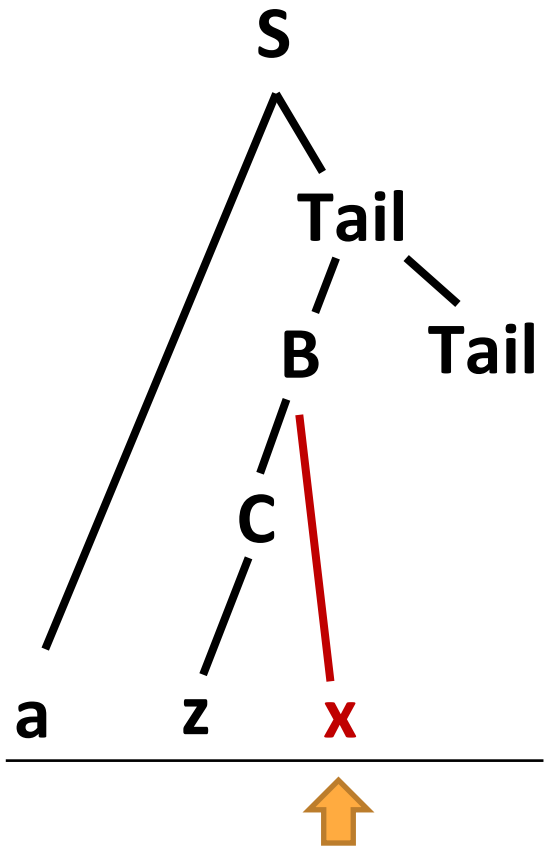
Top-Down Derivation of “a z x”




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

Lookahead	Remaining
z	x

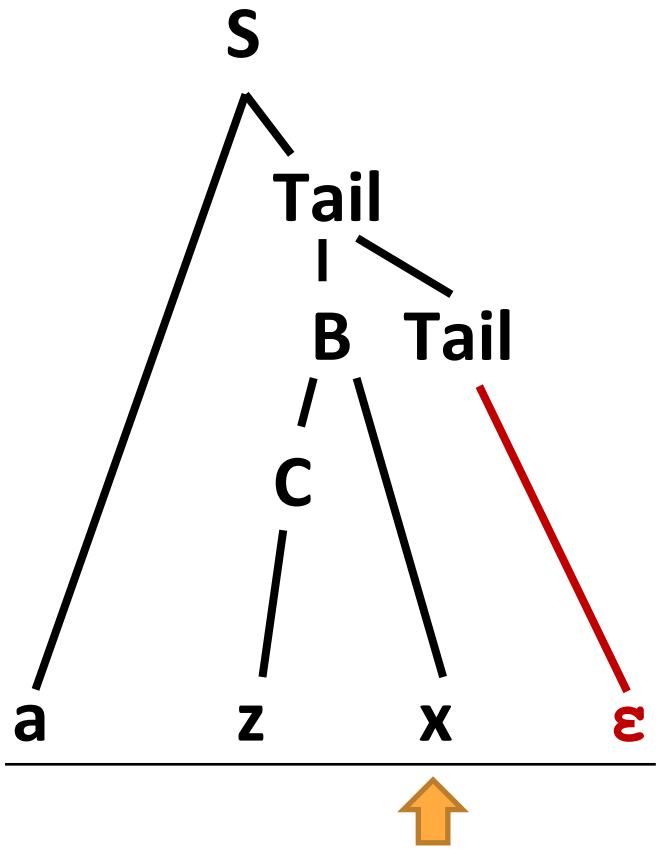
Top-Down Derivation of “a z x”




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

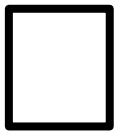
Lookahead Remaining

Top-Down Derivation of “a z x”

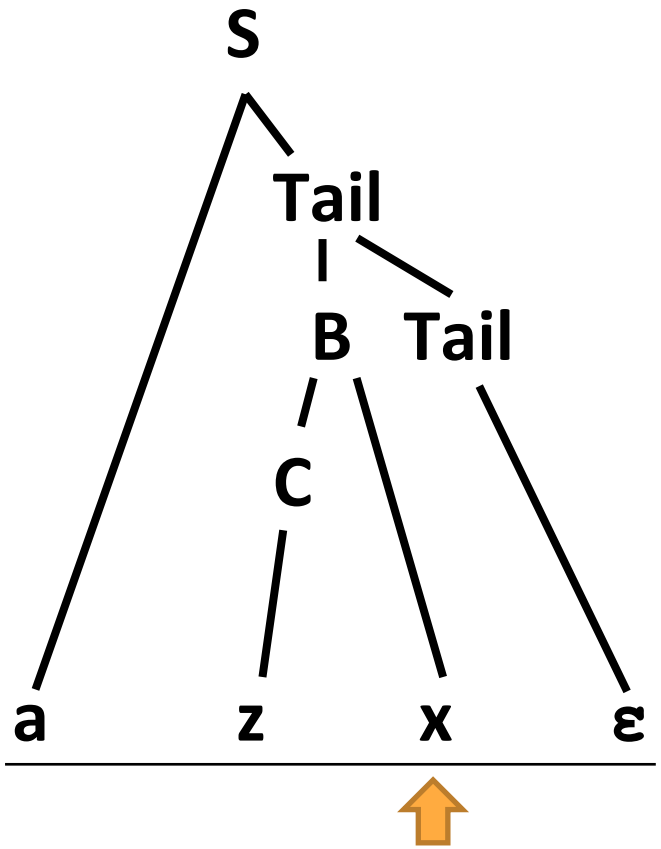



```
0. S ::= a Tail | w Tail
1. Tail ::= B Tail | ε
2. B ::= C x | y
3. C ::= ε | z
```

Lookahead Remaining



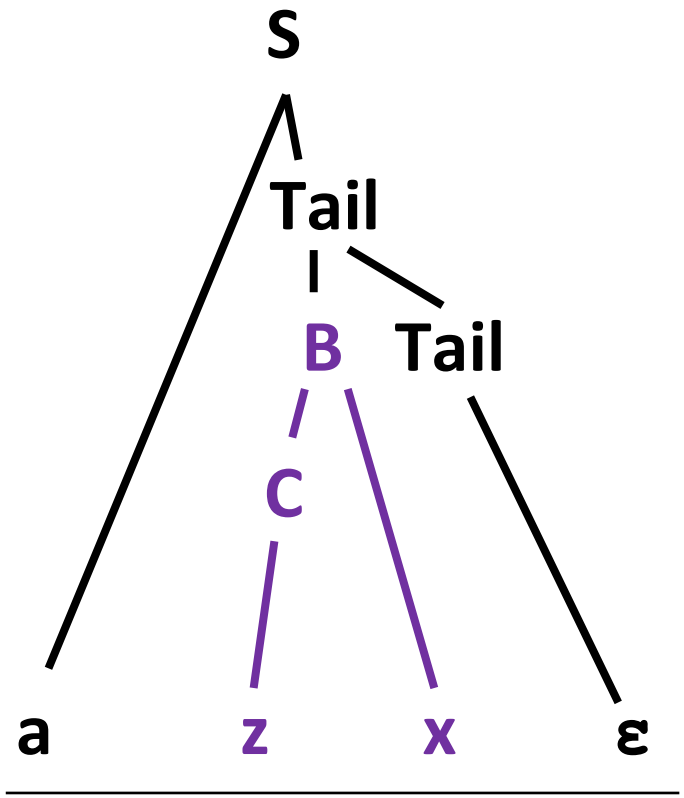
Top-Down Derivation of “a z x”



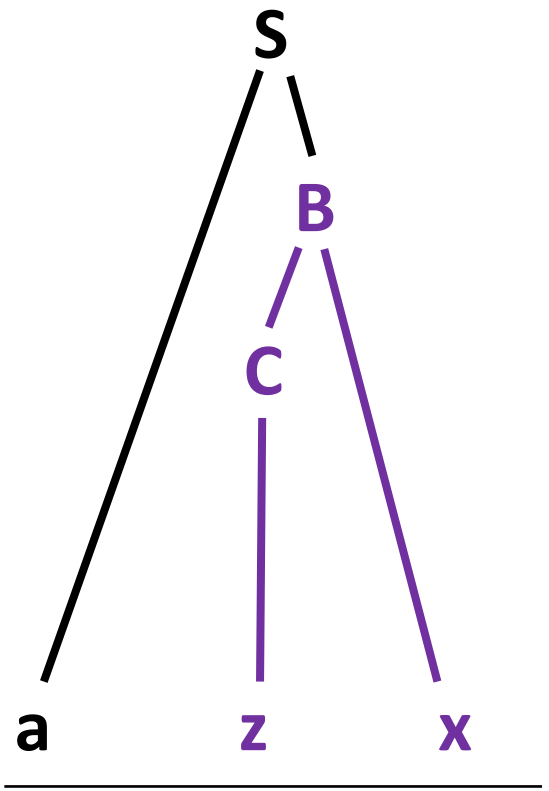
0. **S ::= a Tail | w Tail**
1. **Tail ::= B Tail | ε**
2. B ::= C x | y
3. C ::= ε | z

Success!

Top-Down Derivation of “a z x”



Purple trees are the same again!



Agenda

- What LL (Top-Down) Parsing Looks Like
- LL Grammar Issues
  - FIRST Conflict
  - FIRST FOLLOW Conflict
  - Left Recursion
  - **Indirect Left Recursion**
- Interpreters vs Compilers

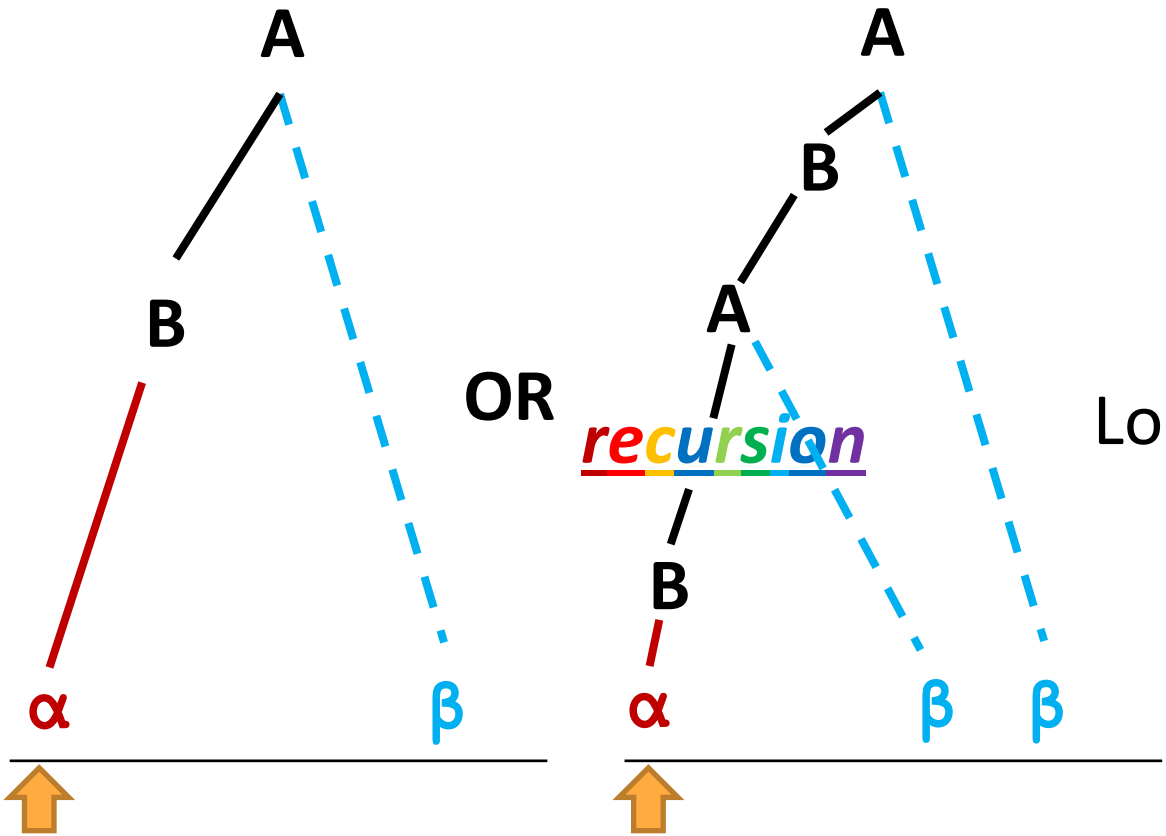
Canonical Indirect Left Recursion Problem

Problem

```
0. A ::= B β
1. B ::= A | α
```

Indirect Left recursion can't be parsed by LL(1) parsers either

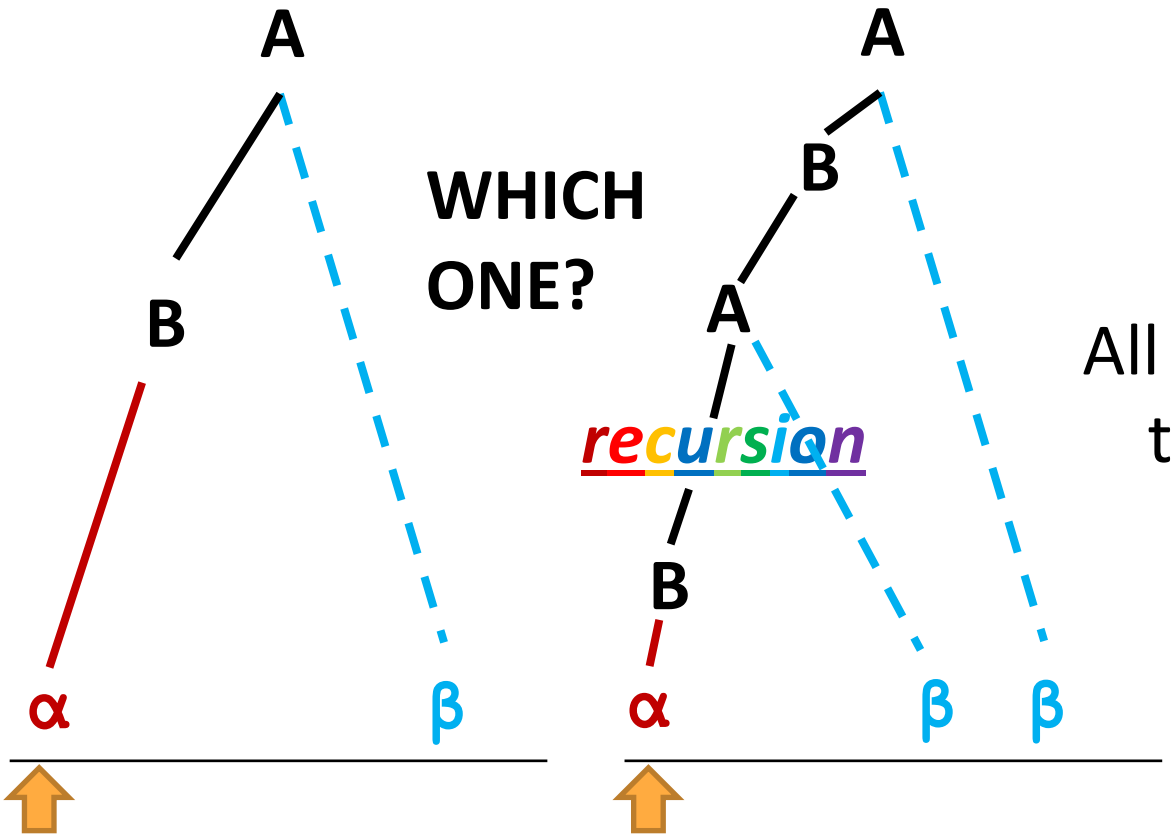
Let's try a top-down derivation of "αβ"




```
0. A ::= B β
1. B ::= A | α
```

Lookahead	Remaining
α	β

Let's try a top-down derivation of "αβ"




```
0. A ::= B β
1. B ::= A | α
```

All we know right now is that it starts with **α!**

Canonical Indirect Left Recursion Solution

Solution

```
0. A ::= B β
1. B ::= A | α
```

Substitute B into A

```
0. A ::= A β | α β
```

Fix via typical Left Recursion Solution

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

Note: **NEVER** remove the starting non-terminal!
The start symbol of the new grammar should be the same

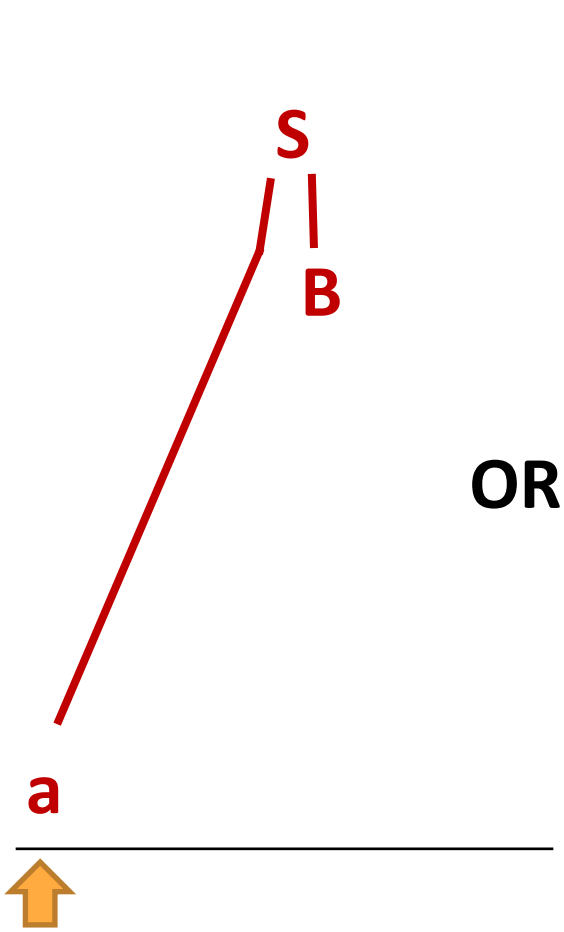
Indirect Left Recursion

Changing the grammar for the last time…

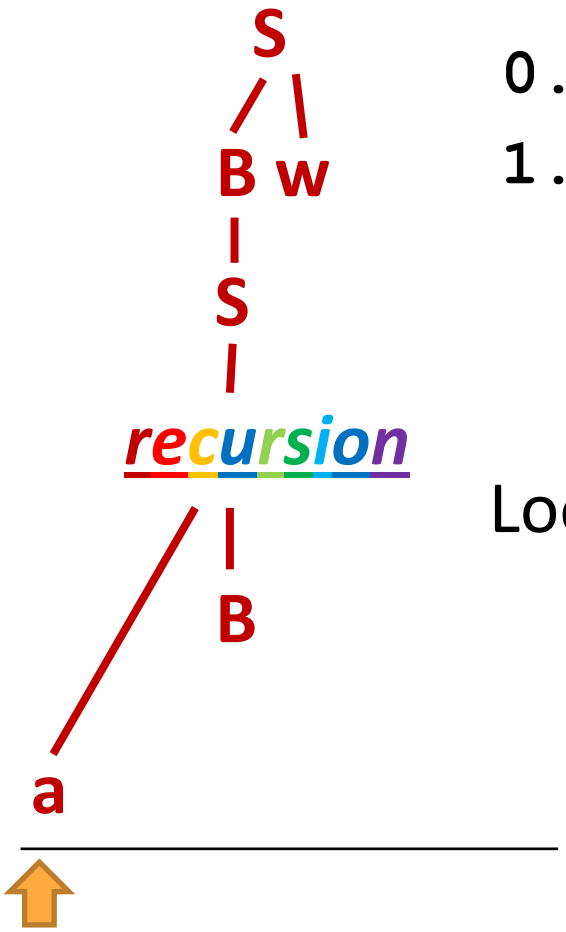
```
0. S ::= B w | a B
1. B ::= S | z x
```

Lookahead	Remaining
a	z x

Indirect Left Recursion



OR




```
0. S ::= B w | a B
1. B ::= S | z x
```

Lookahead	Remaining
a	z x

Applying the fix: Substitute, then eliminate left recursion!

1

0. S ::= S w | z x w | a S | a z x

~~1. B ::= S | z x~~

2

0. S ::= z x w T | a S T | a z x T

1. T ::= w T | ε

3

0. S ::= z x w T | a aT

1. aT ::= S T | z x T

2. T ::= w T | ε

Something else is wrong!

What is it?

Eliminating FIRST CONFLICT

4: sub S into the S in aT

0. S ::= z x w T | a aT
1. aT ::= z x w T T | a aT T | z x T
2. T ::= w T | ε

5: Add a tail to zx as it appears twice in (4)

0. S ::= z x w T | a aT
1. aT ::= z x zT| a aT T
2. zT ::= w T T | T
3. T ::= w T | ε

Eliminating FIRST CONFLICT

6: Sub rule 3 into the T in rule 2.

0. S ::= z x w T | a aT
1. aT ::= z x zT| a aT T
2. zT ::= w T T | w T | ε
3. T ::= w T | ε

7: Notice in rule 2, w T T is the same as w T, since it's recursively appending w's on the right. Get rid of w T T.

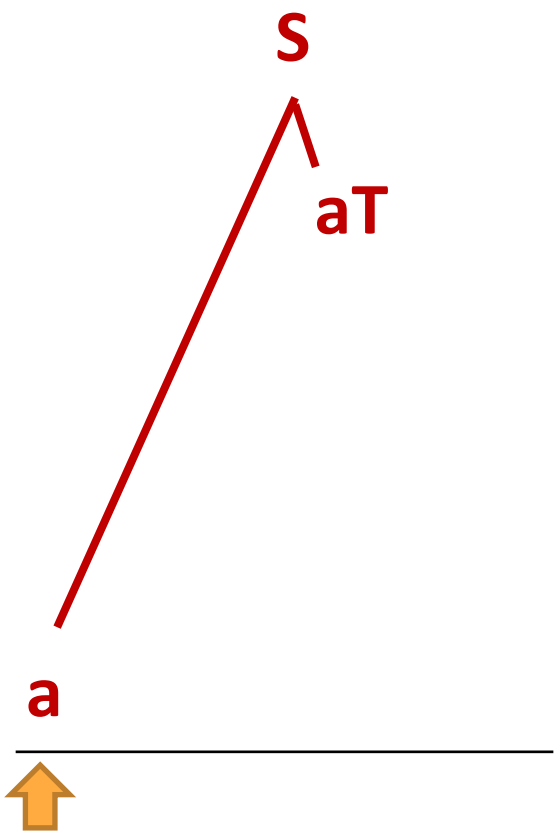
0. S ::= z x w T | a aT
1. aT ::= z x zT| a aT T
2. zT ::= w T | ε
3. T ::= w T | ε

Eliminating FIRST CONFLICT

8: zT has been deleted because it was the same production as T. zT was replaced by T.

0. S ::= z x w T | a aT
1. aT ::= z x T | a aT T
2. T ::= w T | ε

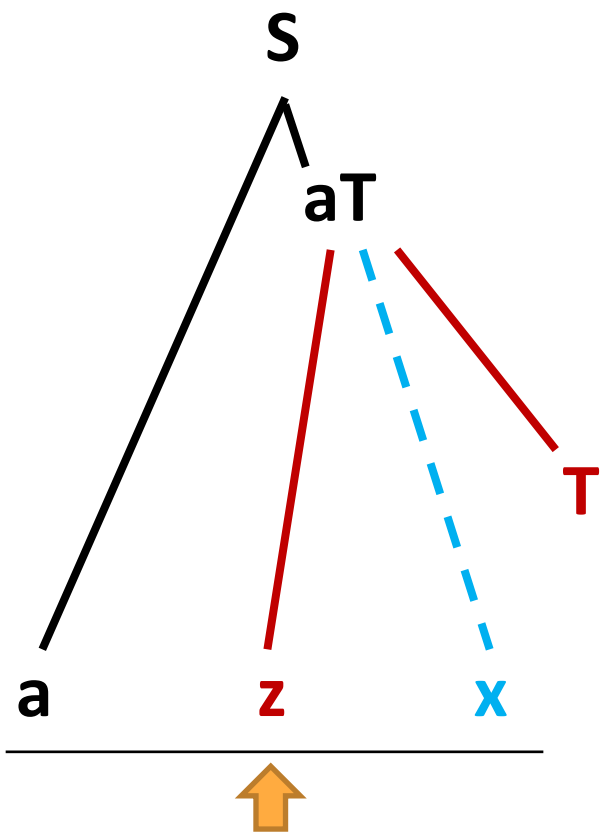
Top-down derivation of “azx”




```
0. S ::= z x w T | a aT
1. aT ::= z x T | a aT T
2. T ::= w T | ε
```

Lookahead	Remaining
a	z x

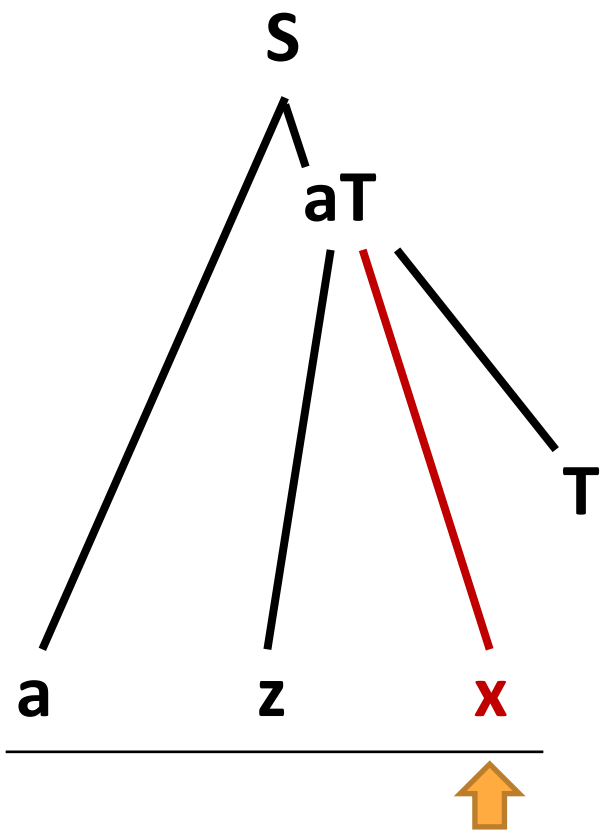
Top-down derivation of “azx”




```
0. S ::= z x w T | a aT
1. aT ::= z x T | a aT T
2. T ::= w T | ε
```

Lookahead	Remaining
z	x

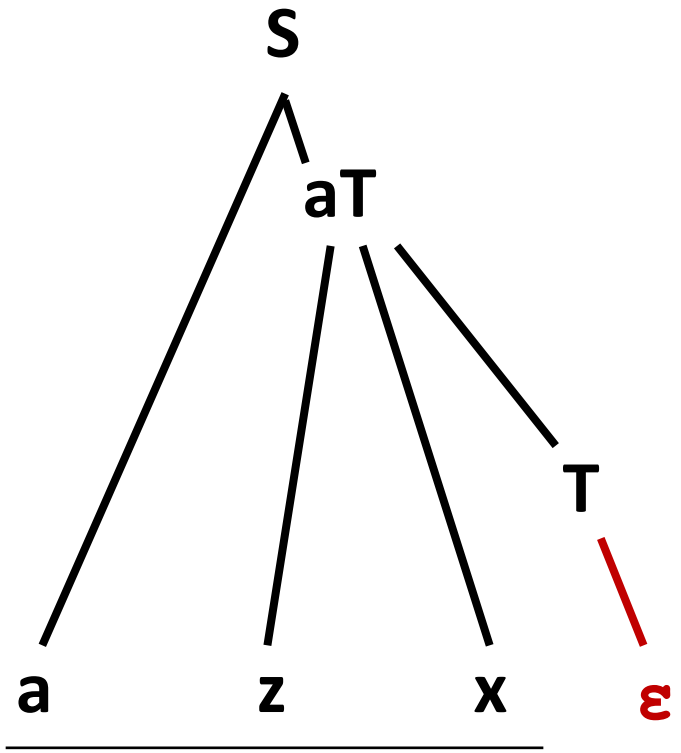
Top-down derivation of “azx”




```
0. S ::= z x w T | a aT
1. aT ::= z x T | a aT T
2. T ::= w T | ε
```

Lookahead Remaining

Top-down derivation of “azx”




```
0. S ::= z x w T | a aT
1. aT ::= z x T | a aT T
2. T ::= w T | ε
```

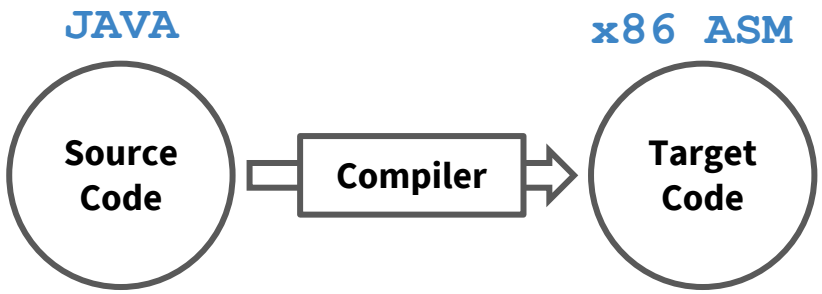
Success!

Agenda

- What LL (Top-Down) Parsing Looks Like
- LL Grammar Issues
  - FIRST Conflict
  - FIRST FOLLOW Conflict
  - Left Recursion
  - Indirect Left Recursion
- **Interpreters vs Compilers**

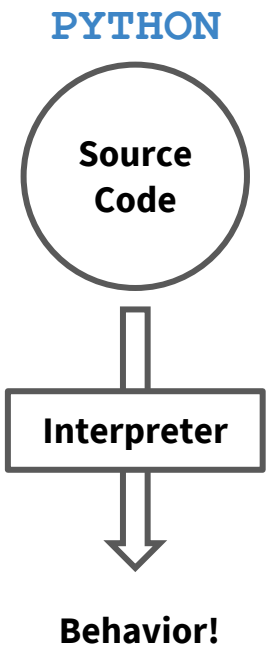
Interpreters vs. Compilers

- Compilers
  - Translate between different languages
  - e.g. MiniJava ⇒ x86 ASM
  - e.g. Java ⇒ Java Byte Code



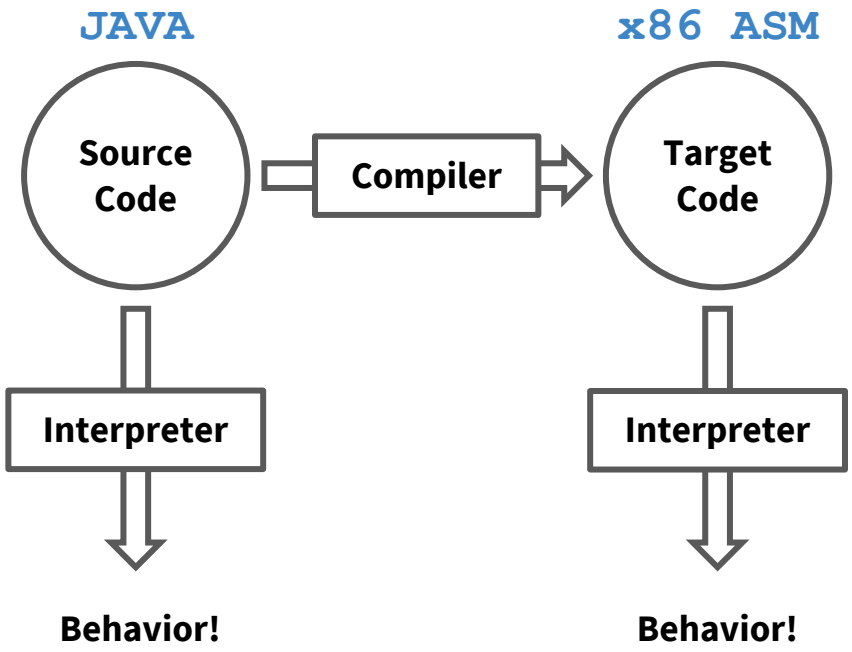
Interpreters vs. Compilers

- Compilers
  - Translate between different languages
  - e.g. MiniJava ⇒ x86 ASM
  - e.g. Java ⇒ Java Byte Code
- Interpreters
  - Take action upon a piece of code as it is read



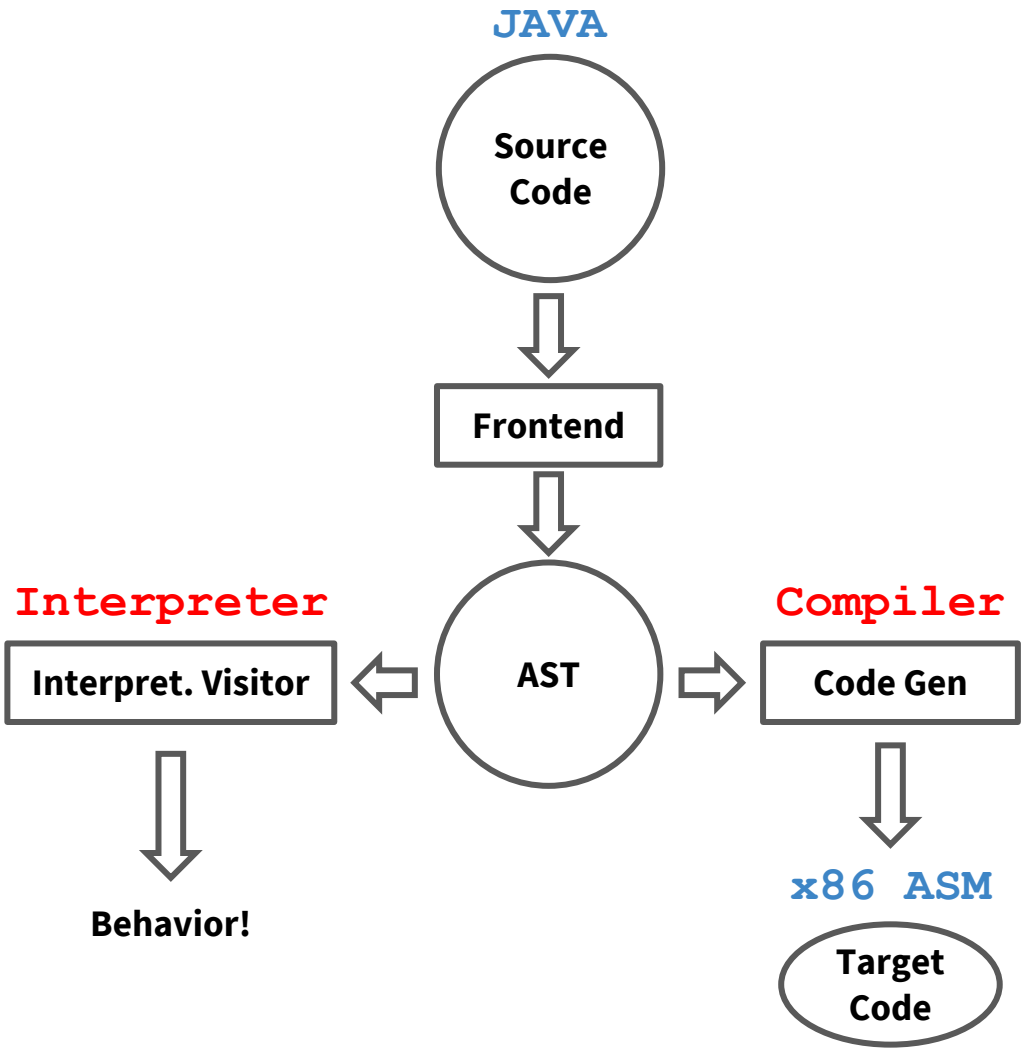
Interpreters vs. Compilers

- Compilers
  - Translate between different languages
  - e.g. MiniJava ⇒ x86 ASM
  - e.g. Java ⇒ Java Byte Code
- Interpreters
  - Take action upon a piece of code as it is read



Implementation

- Frontend is the same!
- Interpreters
  - Execute the AST
- Compilers
  - Translate the AST



Interpreter Demo