Interpreters & More LL Grammars

CSE 401 Section 5

Kory Watson Aaron Johnston

Miya Natsuhara Sam Wolfson

Announcements

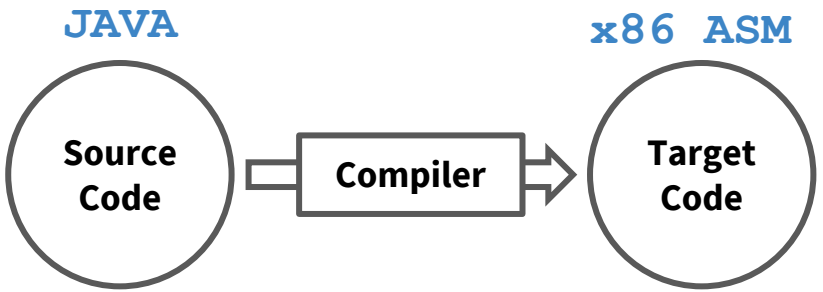
- Parser & AST due ***tonight***!

- Homework 3 (LL grammars) due Monday
  - Only **one** late day max so we can distribute solutions in time for…

- *Midterm next Friday (1 week away)!*

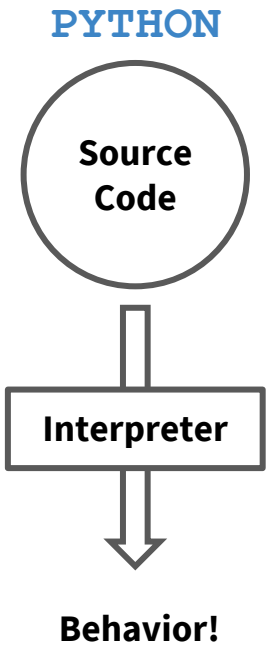
Interpreters vs. Compilers

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



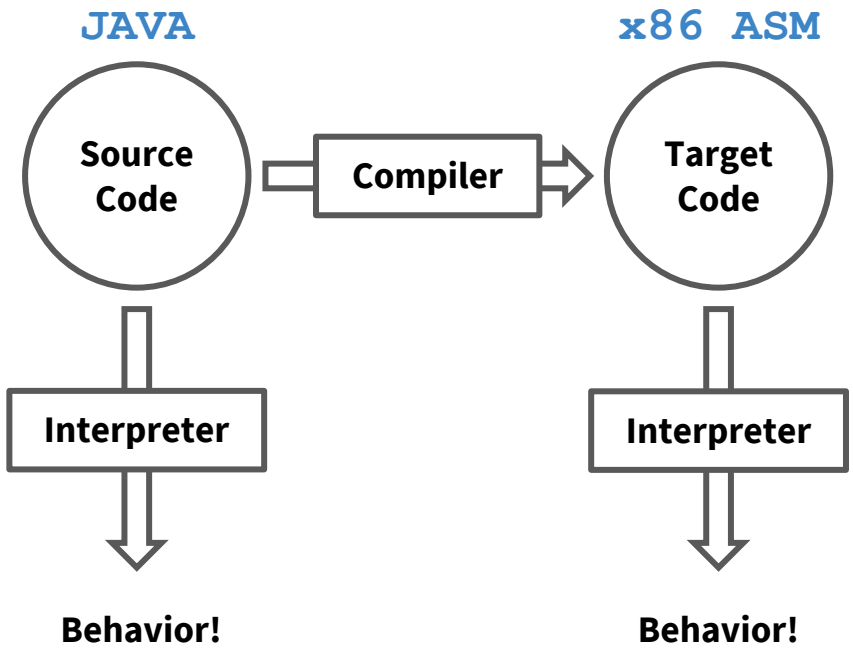
Interpreters vs. Compilers

- 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



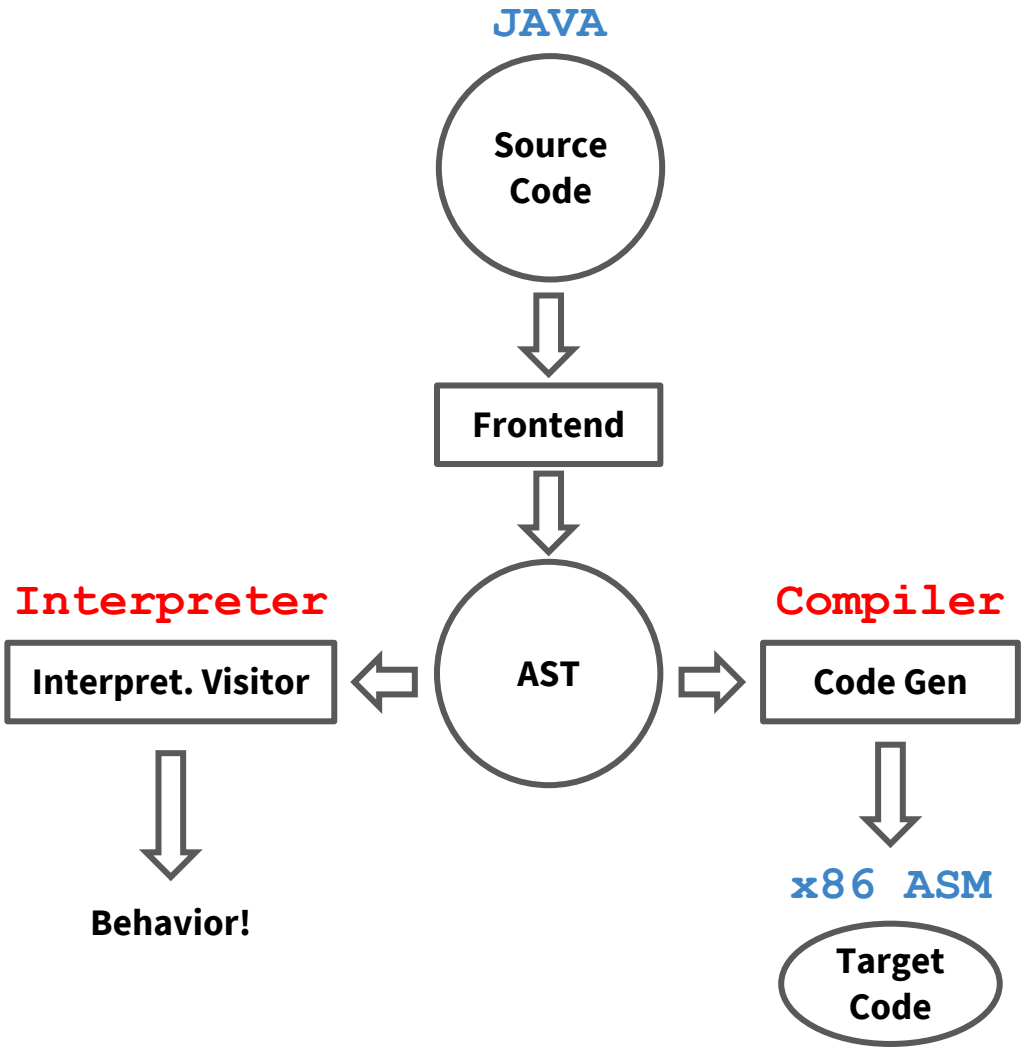
Interpreters vs. Compilers

- 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

LL Grammars

Now that you've taken a look at LL grammars, let's review again…

L L (1)

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

Agenda

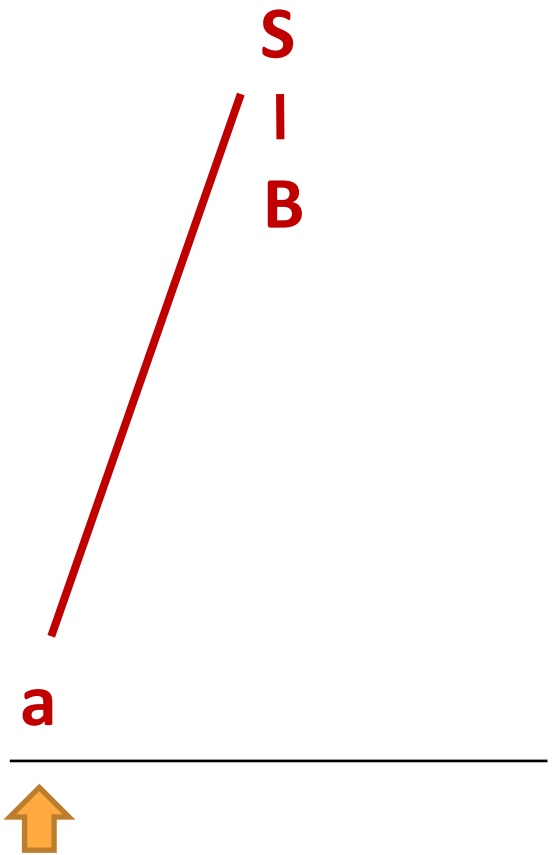
- **What LL (Top-Down) Parsing Looks Like**
- LL Grammar Issues
  - FIRST Conflict
  - FIRST FOLLOW Conflict
  - Recursion
  - Indirect Left Recursion

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**

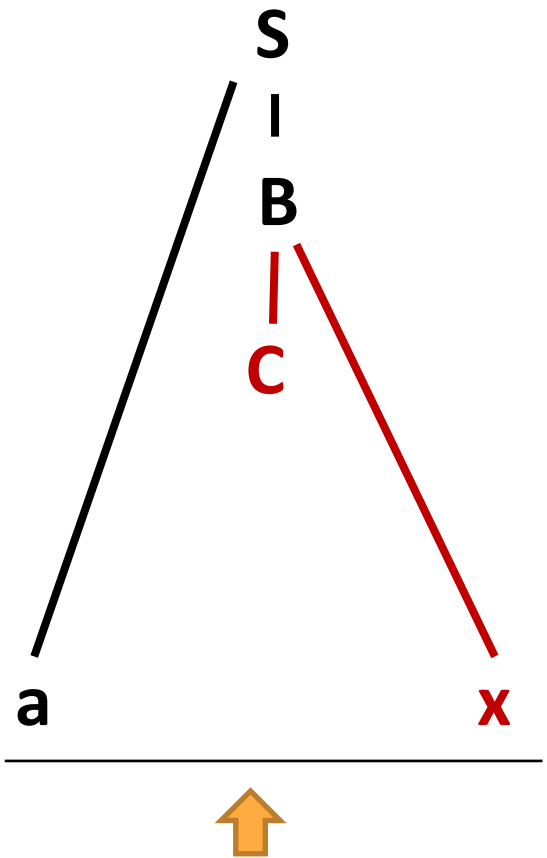
Top-Down Derivation of “a z x”




```
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”

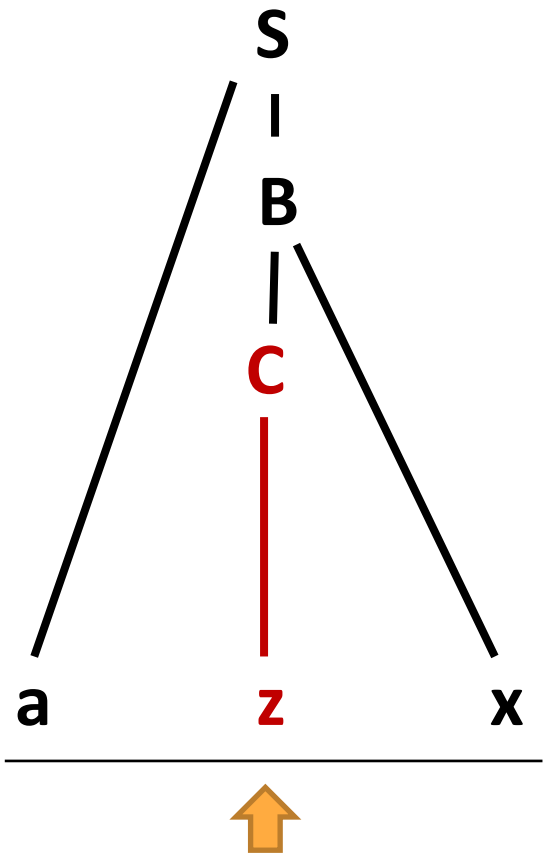



```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | z
```

Lookahead Remaining

z x

Top-Down Derivation of “a z x”



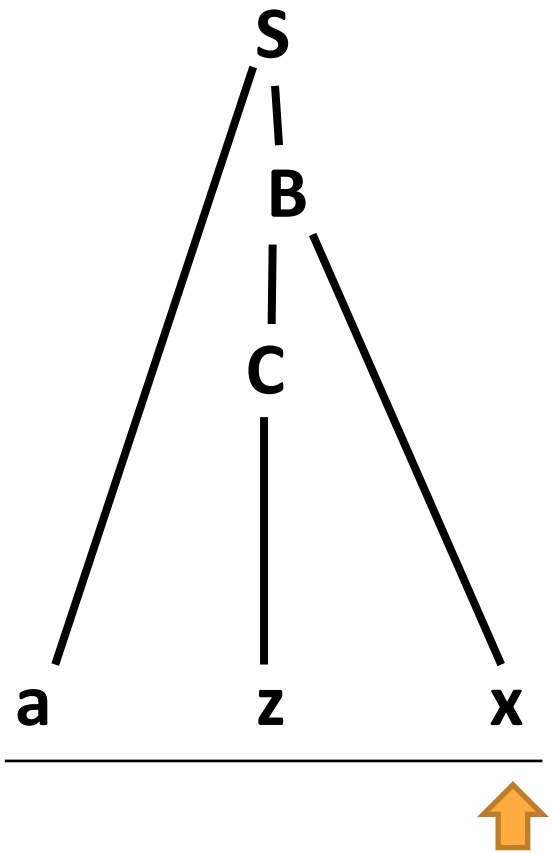

```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | z
```

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

x

Agenda

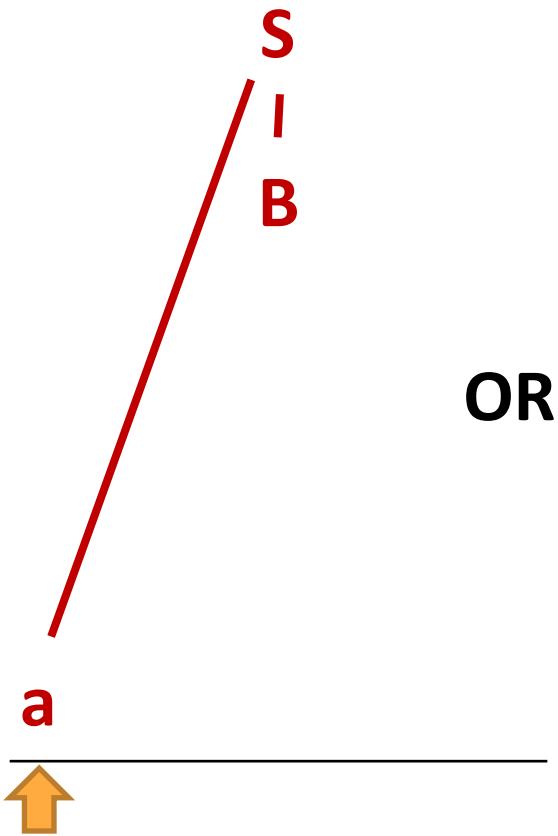
- What LL (Top-Down) Parsing Looks Like
- LL Grammar Issues
  - **FIRST Conflict**
  - FIRST FOLLOW Conflict
  - Recursion
  - Indirect Left Recursion

Let's change the grammar a little bit. (Grammar 1)

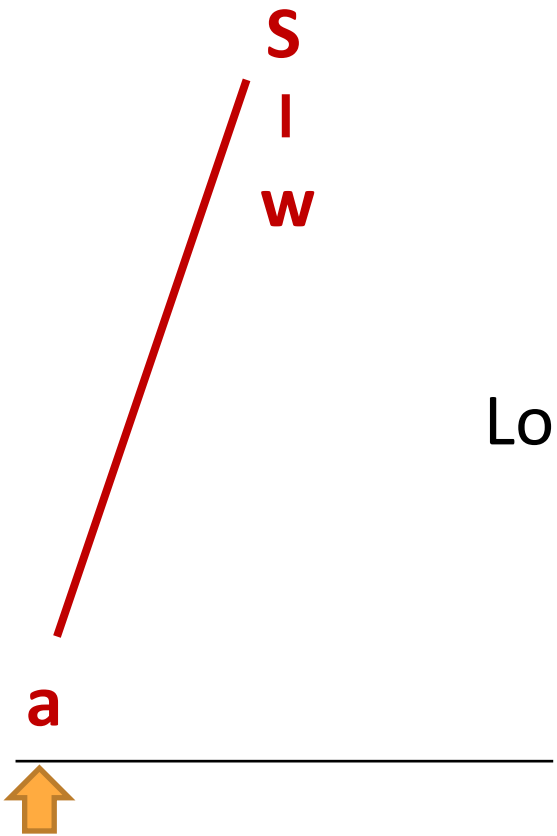
```
0. S ::= a B | a w
1. B ::= C x | y
2. C ::= ε | z
```

Lookahead	Remaining
a	z x

Top-Down Derivation of “a z x”



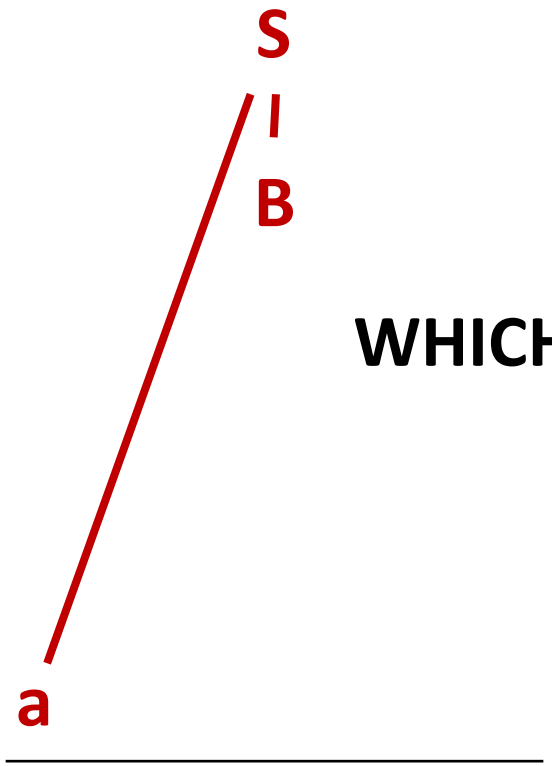
OR



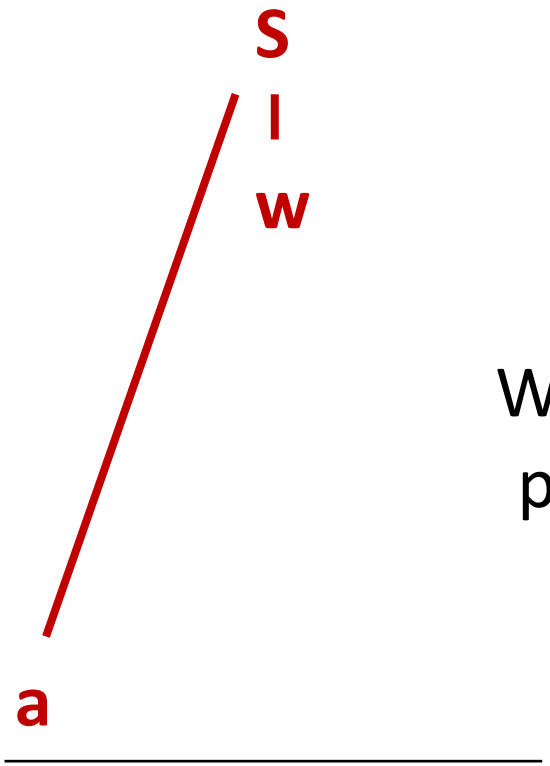

```
0. S ::= a B | a w
1. B ::= C x | y
2. C ::= ε | z
```

Lookahead	Remaining
a	z x

Top-Down Derivation of “a z x”



WHICH?



We don’t know!

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

What's the issue?

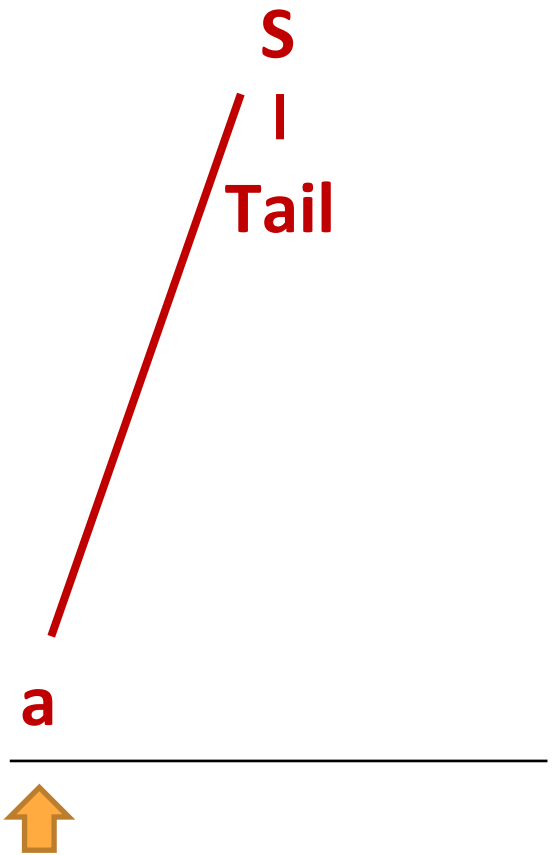
```
0. S ::= a B | a w
1. B ::= C x | y
2. C ::= ε | z
```

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

Fix: Factor out the Common Prefix

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

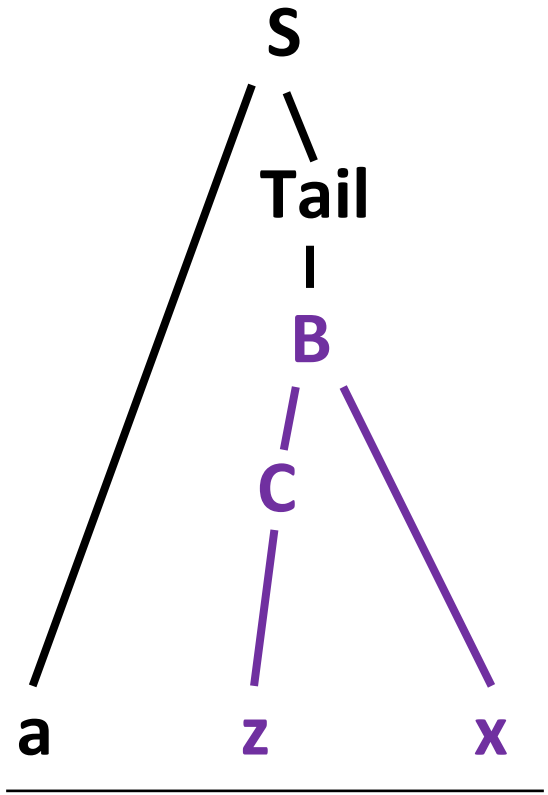
Top-Down Derivation of “a z x”



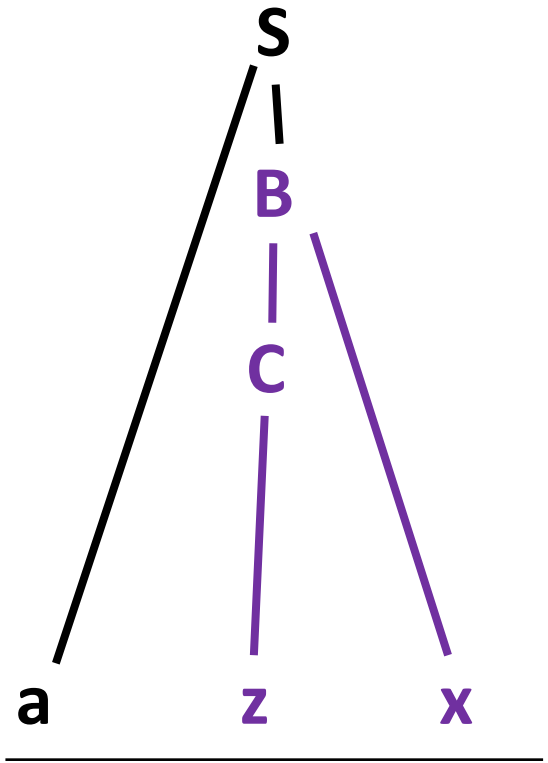

```
0. S ::= a Tail
1. Tail ::= B | w
2. B ::= C x | y
3. C ::= ε | z
```

Lookahead	Remaining
a	z x

Top-Down Derivation of “a z x”



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

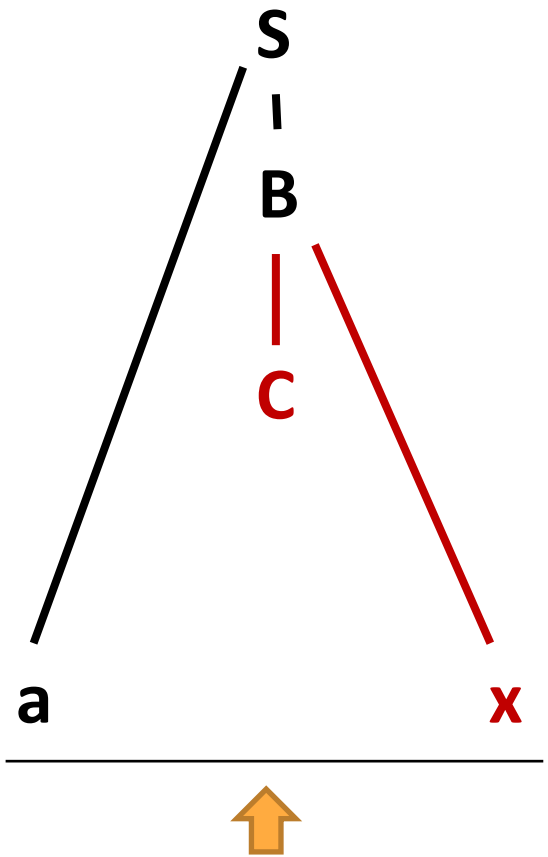
Watch out for empty rhs (ε-productions) too! (Grammar 2)

Changing the grammar again…

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

Lookahead	Remaining
a	x

Watch out for empty rhs (ε-productions) too! (Grammar 2)

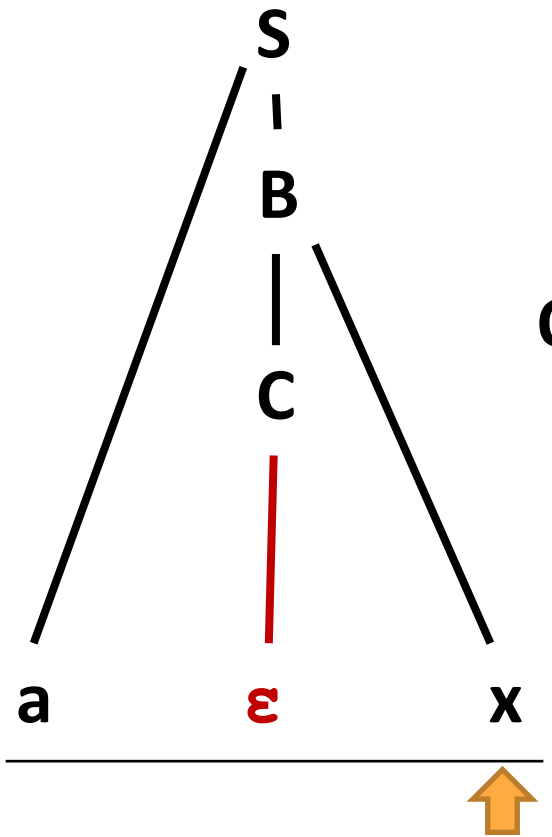



```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | x
```

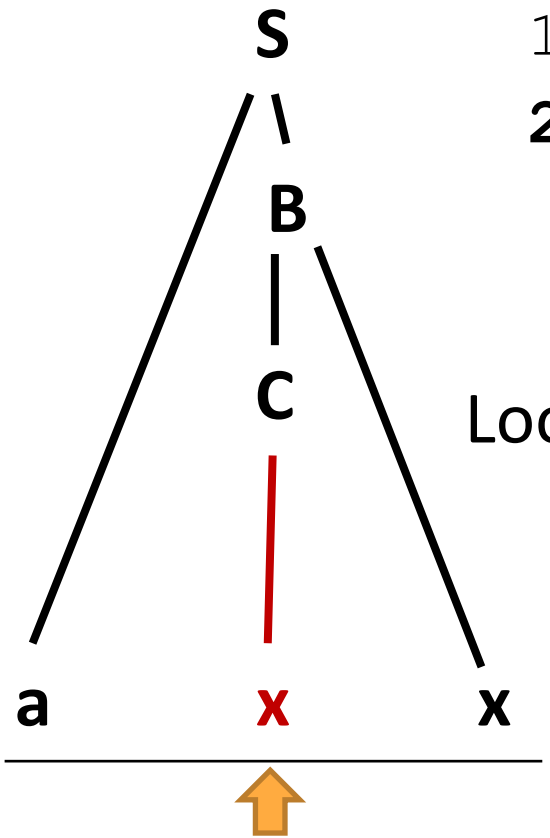
Lookahead Remaining

x

Watch out for empty rhs (ε-productions) too!



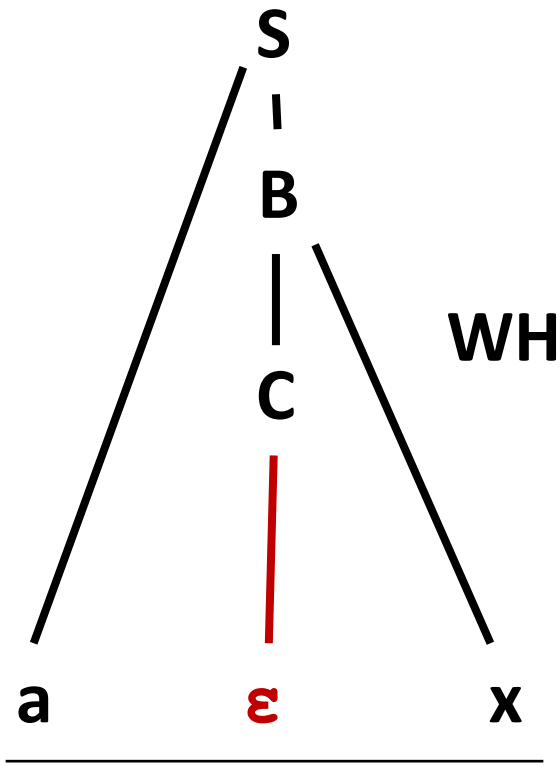
OR



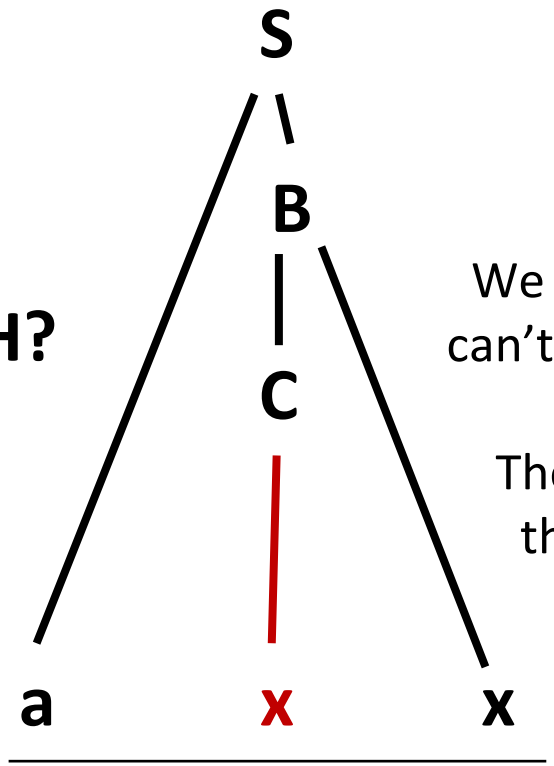

```
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | x
```

Lookahead Remaining

Top-Down Derivation of “a x”



WHICH?



We don’t know!

We are using an *LL(1)* parser, we can’t look at more than **the only *x***!

Therefore, we can’t know that there is no input after the x.

What's the issue?

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

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

Fix (Various): Substitute the Common Prefix, then Factor

1

0. S ::= a B
2. B ::= x | xx | y
~~3. C ::= ε | x~~

2

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

Agenda

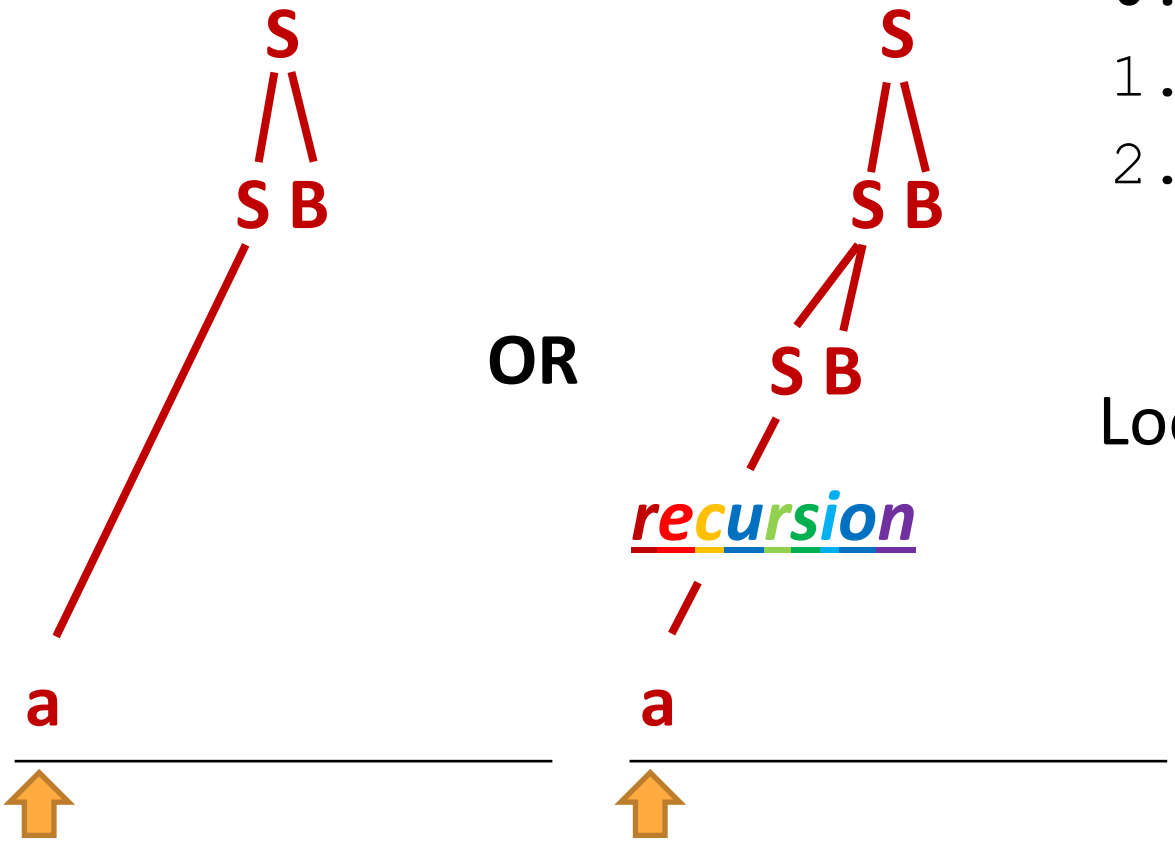
- What LL (Top-Down) Parsing Looks Like
- LL Grammar Issues
  - FIRST Conflict
  - FIRST FOLLOW Conflict
  - **Left Recursion**
  - Indirect Left Recursion

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

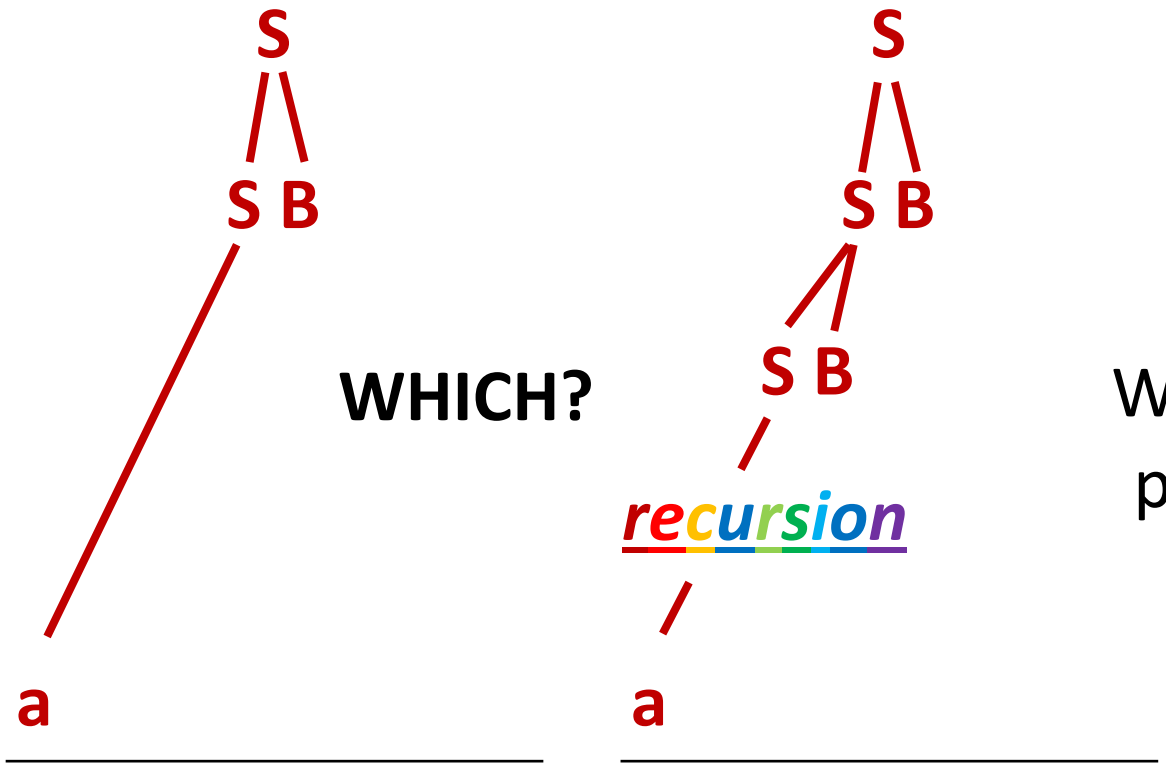
Top-Down Derivation of “a z x”




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

Lookahead	Remaining
a	z x

Top-Down Derivation of “a z x”



We don’t know!

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

What's the issue?

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

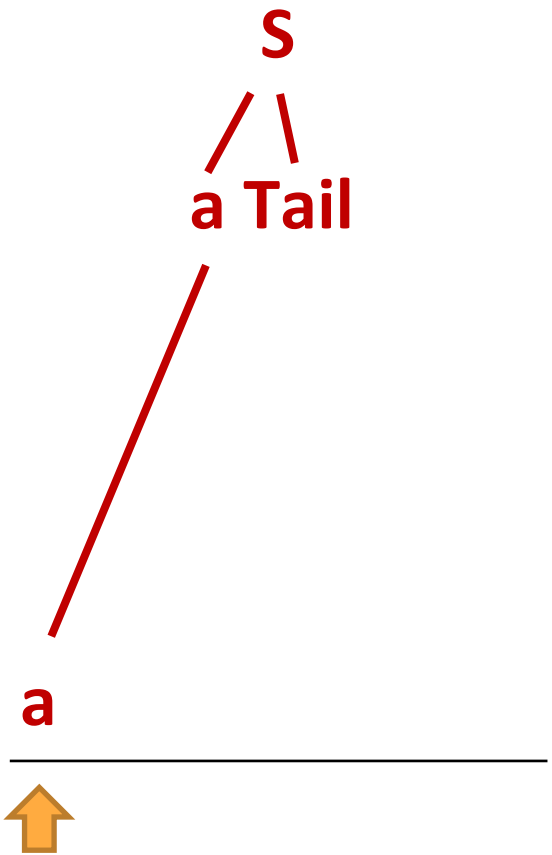
Left recursion can't be parsed by LL(1) parsers!

To fix the issue: Make a tail rule again!

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

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

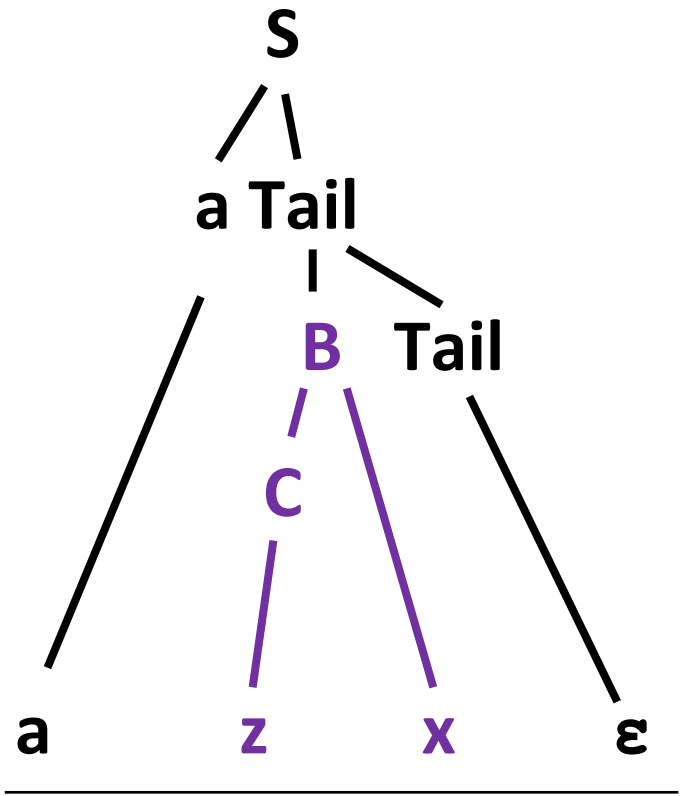
Top-Down Derivation of “a z x”



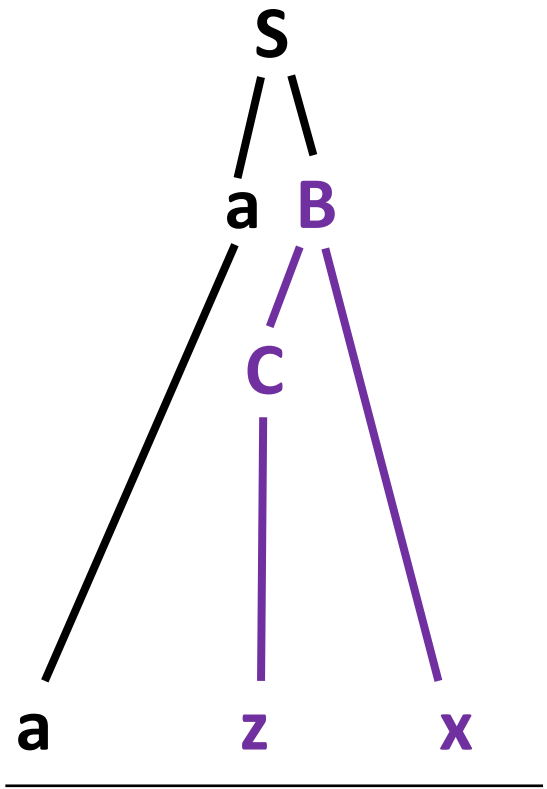
0. **S ::= a Tail | w Tail**
1. **Tail ::= B Tail | ε**
2. B ::= C x | y
3. C ::= ε | z

Lookahead	Remaining
a	**z x**

Top-Down Derivation of “a z x”



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**

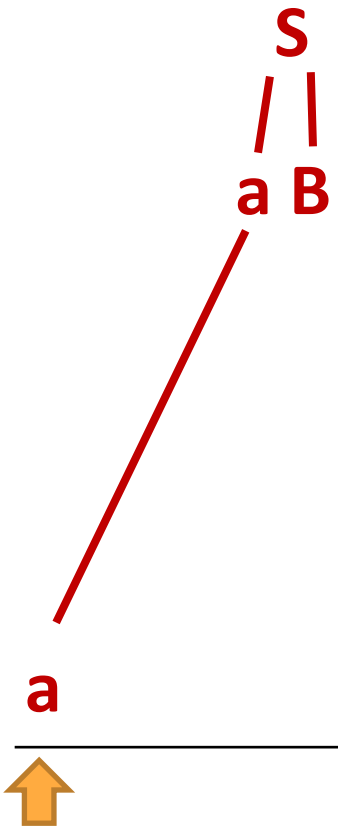
Watch out for indirect left recursion, too!

Changing the grammar again…

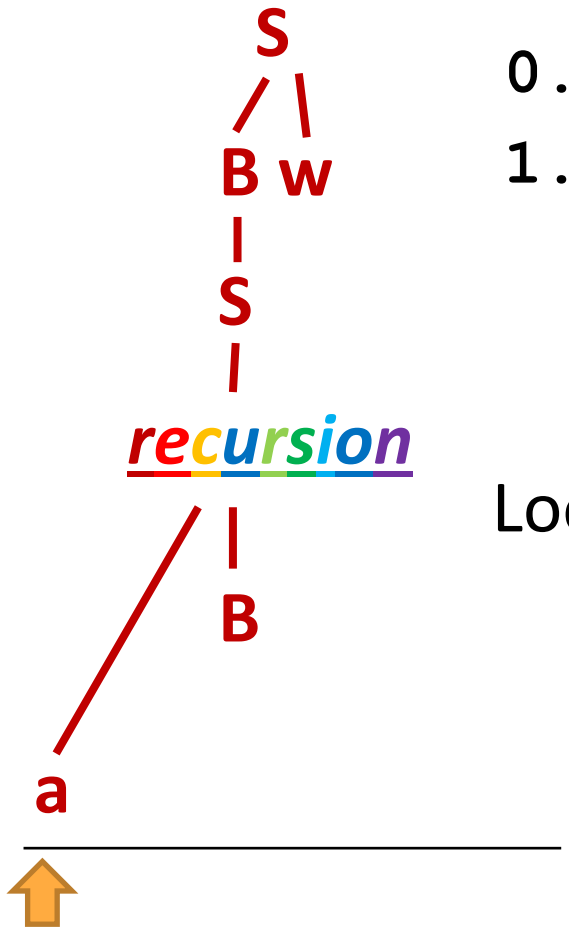
```
0. S ::= B w | a B
1. B ::= S | z x | y
2. C ::= C x | y
```

Lookahead	Remaining
a	z x

Watch out for indirect left recursion, too!



OR




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

Lookahead	Remaining
a	z x

To fix the issue: 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 Tail | a S Tail | a z x Tail
1. Tail ::= w Tail | ε
```

3

```
0. S ::= z x w Tail | a aTail
1. aTail ::= S Tail | z x Tail
2. Tail ::= w Tail | ε
```

Note: **NEVER** remove the starting non-terminal!

The start symbol of the new grammar should be the same