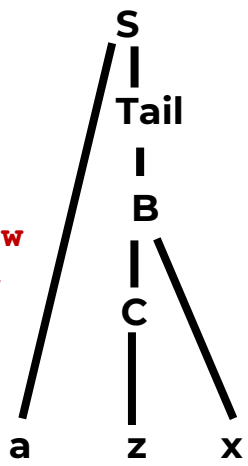
CSE 401 - LL Semantics, Semantics, Type Checking, & Vtables

Edit the following Grammars to make them LL(1). Then walk through the top down parse for the string given in the parenthesis.

Grammar 1 ("azx")

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

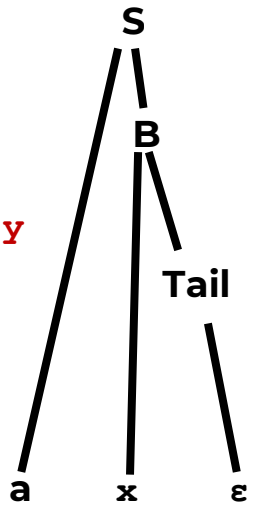
```
0. S ::= a Tail
1. Tail ::= B | w
2. B ::= C x | y
3. C ::= ε | z
```



Grammar 2 ("ax")

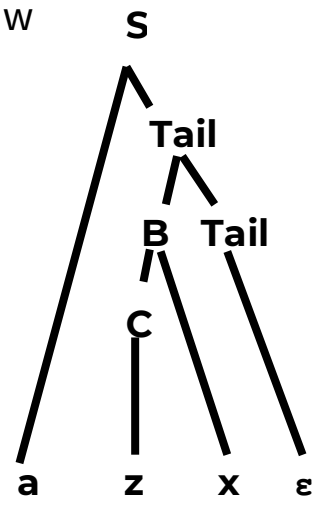
0. S ::= a B
1. B ::= C x | y
2. C ::= ε | x

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



Grammar 3 ("azx")

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




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

Grammar 4 ("axzw")

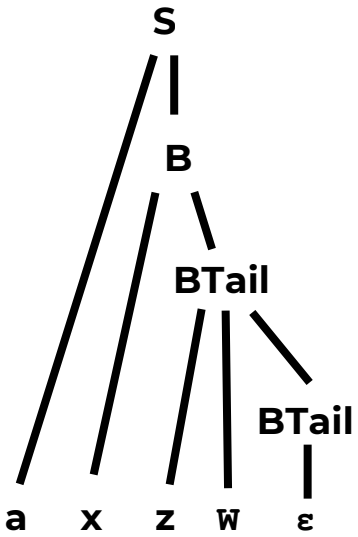
0. S ::= B w | a B
1. B ::= C w | x
2. C ::= B z | ε

First, substitute C into B to eliminate the indirect left recursion.

0. S -> B w | a B
1. B -> B z w | w | x

Add "BTail" nonterminal to Handle direct left recursion for B.

0. S ::= B w | a B
1. B ::= w BTail | x BTail
2. BTail ::= z w BTail | ε



2. Suppose we have the following global scope:

```
class Bar { boolean field; public int method(int i, int j); }
class Foo extends Bar { int val; public boolean whoop(int x); }
```

Now, consider the following hypothetical method definition for `Bar.method`:

```
public int method(int i, int j) {
  int r;
  boolean b;
  Foo o;
  if (this.field) {
    o = this;
    b = o.whoop(i + j);
    r = o.val;
  } else {
    r = i * j + 3;
  }
  return r;
}
```

a. What variables (locals, parameters, etc.) are defined in the *local* scope in the `method` body?

```
Bar this; int i; int j; int r; boolean b; Foo o;
```

Remember that every MiniJava method has an implicit parameter "**`this`**" for the receiver object. For the sake of type-checking the method body, it makes sense to treat it like a normal parameter, although you may treat it however you'd like in your symbol tables.

b. When we execute this method body, a runtime error could result. Explain how something could go wrong by giving values of the parameters and/or variables involved that would cause a runtime error.

```
this = Bar(field: true);
```

The error here is the potential failure of the downcast in the assignment "`o = this`." Similar to real Java, MiniJava does not support implicit downcasts, so it is possible that this statement results in an incompatible assignment error.

c. The method body also has type errors. Can you describe which type check(s) the compiler could use to deduce this fact?

   Since MiniJava's static semantics do not support downcasts, a MiniJava compiler must check that the type of an assignment statement's right-hand side is either the same as the left-hand side's type or a subclass type of the left-hand side's class type.

d. Does *every* possible execution of this method produce a runtime error? Can you describe any that happen to be statically correct? (Again, possible runtime values for parameters/variables would suffice.)

   No, some possible executions of the method avoid the branch that causes an issue, for example given the following value of **`this`**:

   ```
   this = Bar(field: false);
   ```

   Alternatively, some possible executions could enable the "downcast" to succeed, if the receiver object (**`this`**) ends up really being an instance of the subclass **`Foo`**, like so:

   ```
   this = Foo(field: true, val: <any integer>);
   ```

e. Suppose that we replaced the use of **`this`**`.field` in the method body to call a boolean method that always returns false. How would this change your answers to the previous questions?

   Even though the ill-behaving branch would never get run, type checking is concerned with the *types* of expression and ignores their values. A type checker for MiniJava will verify the **if** body (*i.e.*, will report a type error), despite the forbidden behavior being impossible according to the dynamic semantics of the program. In other words, even if **`this`**`.field` was always false, a type checker for MiniJava would still check that the **if** body followed MiniJava's statics semantics.