

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.

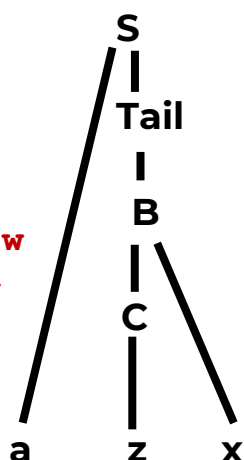
Grammar 1 (“azx”)

0. $S ::= a B \mid a w$

1. $B ::= C x \mid y$

2. $C ::= \epsilon \mid z$

0. $S ::= a \text{ Tail}$
 1. $\text{Tail} ::= B \mid w$
 2. $B ::= C x \mid y$
 3. $C ::= \epsilon \mid z$



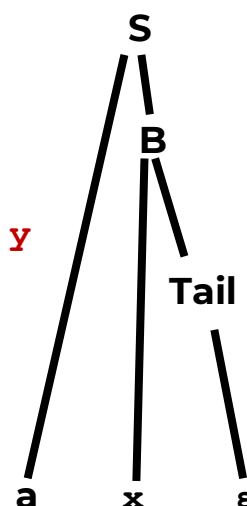
Grammar 2 (“ax”)

0. $S ::= a B$

1. $B ::= C x \mid y$

2. $C ::= \epsilon \mid x$

0. $S ::= a B$
 2. $B ::= x \text{ Tail} \mid y$
 3. $\text{Tail} ::= x \mid \epsilon$

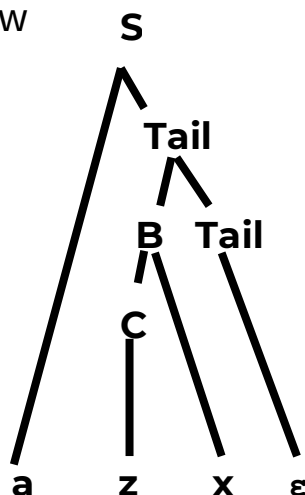


Grammar 3 (“azx”)

0. $S ::= S B \mid a \mid w$

1. $B ::= C x \mid y$

2. $C ::= \epsilon \mid z$



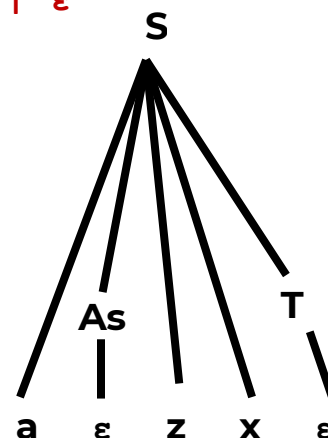
0. $S ::= a \text{ Tail} \mid w \text{ Tail}$
 1. $\text{Tail} ::= B \text{ Tail} \mid \epsilon$
 2. $B ::= C x \mid y$
 3. $C ::= \epsilon \mid z$

Grammar 4 (“azx”)

0. $S ::= B w \mid a B$

1. $B ::= S \mid z x$

0. $S ::= z x w T \mid a A s z x T$
 1. $A s ::= a A s \mid \epsilon$
 2. $T ::= w T \mid \epsilon$



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.`” Unlike real Java, MiniJava’s dynamic semantics defines no behavior for a failing downcast, so the static semantics forbids downcasts altogether.

- 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 forbids 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 composes through types and type signatures (*not* the specific values!), so 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.