# EECS 368
# Programming Language Paradigms

Dr. Andy Gill

Department of Electrical Engineering & Computer Science
University of Kansas

August 24, 2011

KU

# Summary of Last Class

**Syntax trees are a fundamental data structure used to understand and implement computer languages**

**Backus-Naur Form (BNF) is way of expressing valid concrete syntax trees, or grammars**

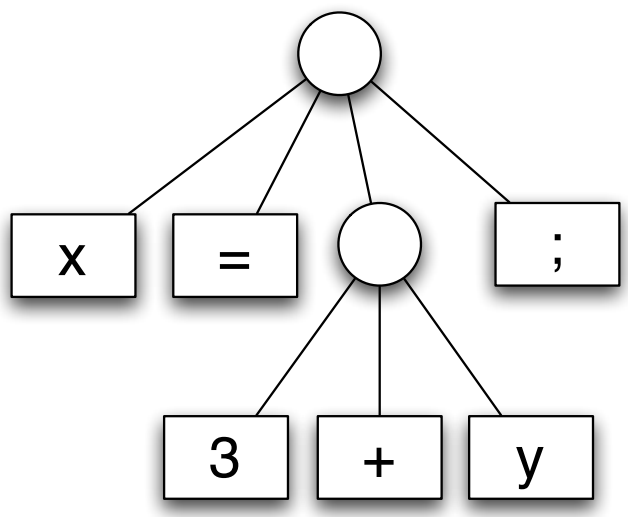
**BNF can be used to guide turning a token stream into a syntax tree**

KU

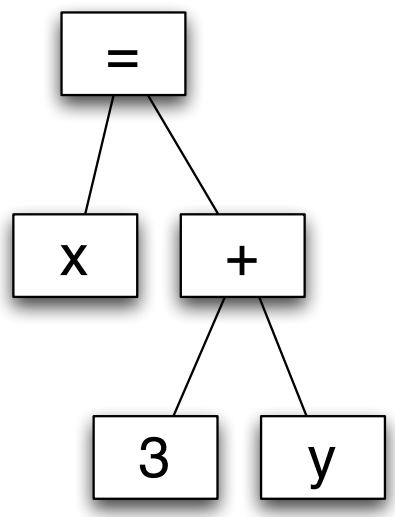
# A Syntax Tree

A tree that represents the syntactical structure of a specific program written in a specific language.

$$x = 3 + y;$$

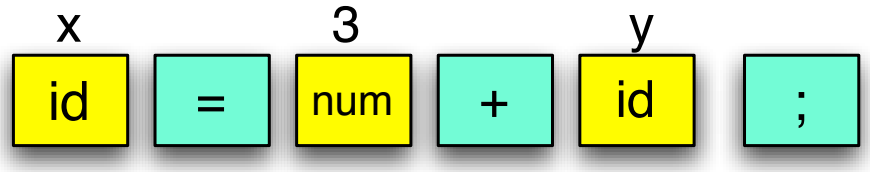



Concrete Syntax Tree




Abstract Syntax Tree

# Finding Structure




$S \rightarrow \texttt{id} = E\ ;$

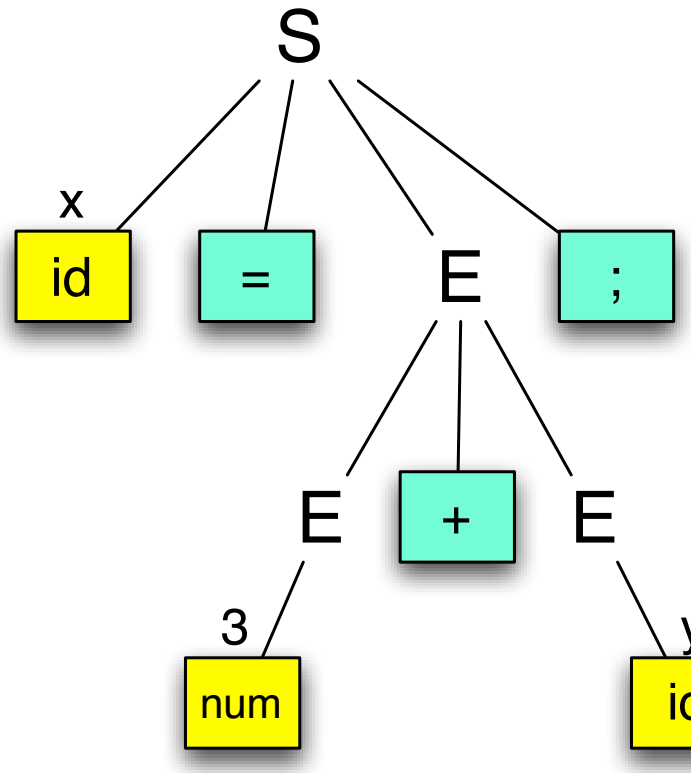
$E \rightarrow \texttt{id}$

$E \rightarrow \texttt{num}$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow ( E )$



# Class Narrative

**In this class we will invent the BNFs for part of Java.**

- Get a feel for how to use BNFs
- Learn some idioms for using BNFs

KU

# Java Program

## Java Example

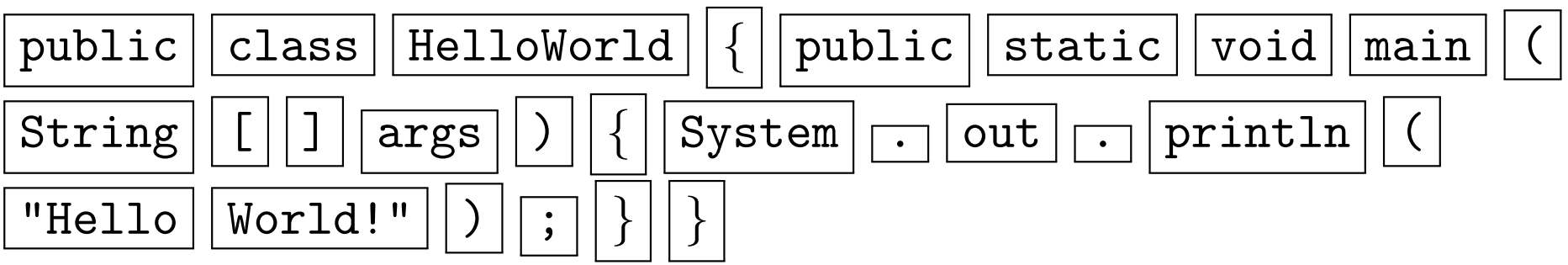
```
public class HelloWorld {
    public static void main (String[] args) {
        System.out.println ("Hello World!");
    }
}
```

```
public class HelloWorld { public static
void main ( String [ ] args ) { System .
out . println ( "Hello World!" ) ; } }
```

# Example Program

## Java Example

```
public class HelloWorld {
    public static void main (String[] args) {
        System.out.println ("Hello World!");
    }
}
```



# First cut Grammar for Java

```
java ::= public class name { method }
```

# First cut Grammar for Java

```
java ::= public class name { method }

method ::= public static void name ( method_arg ) { statement ; }
```

# First cut Grammar for Java

```
java ::= public class name { method }

method ::= public static void name ( method_arg ) { statement ; }

method_arg ::= type name

type ::= String [ ]
```

KU

# First cut Grammar for Java

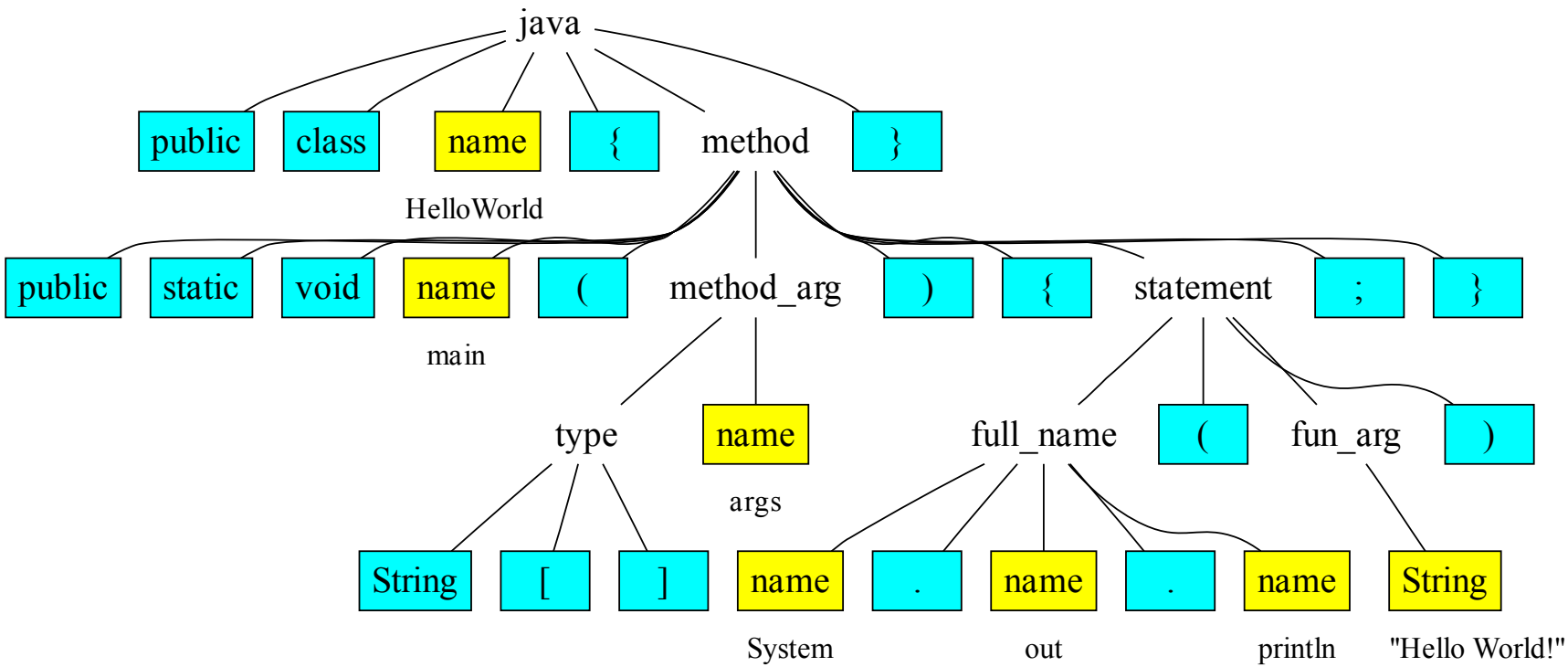
```
java ::= public class name { method }

method ::= public static void name ( method_arg ) { statement ; }

method_arg ::= type name

type ::= String [ ]

statement ::= full_name ( fun_arg )
```

# First cut Grammar for Java

```
java ::= public class name { method }

method ::= public static void name ( method_arg ) { statement ; }

method_arg ::= type name

type ::= String [ ]

statement ::= full_name ( fun_arg )

full_name ::= name . name . name
```

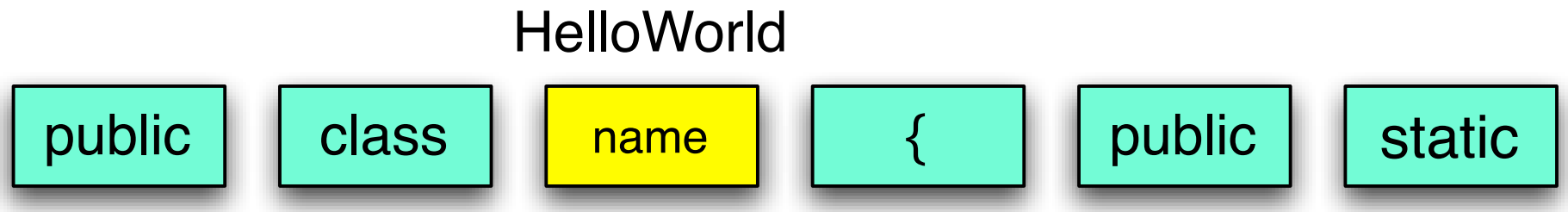
# First cut Grammar for Java

```
java ::= public class name { method }

method ::= public static void name ( method_arg ) { statement ; }

method_arg ::= type name

type ::= String [ ]

statement ::= full_name ( fun_arg )

full_name ::= name . name . name

fun_arg ::= string
```


java
public
class
name
HelloWorld
{
method
}
public
static
void
name
main
(
method_arg
)
{
statement
;
}
type
name
args
full_name
(
fun_arg
)
String
[
]
name
System
.
name
out
.
name
println
String
"Hello World!"

# Keywords

Some names were recognized as independent non-terminals,
some names were recognized as regular names.



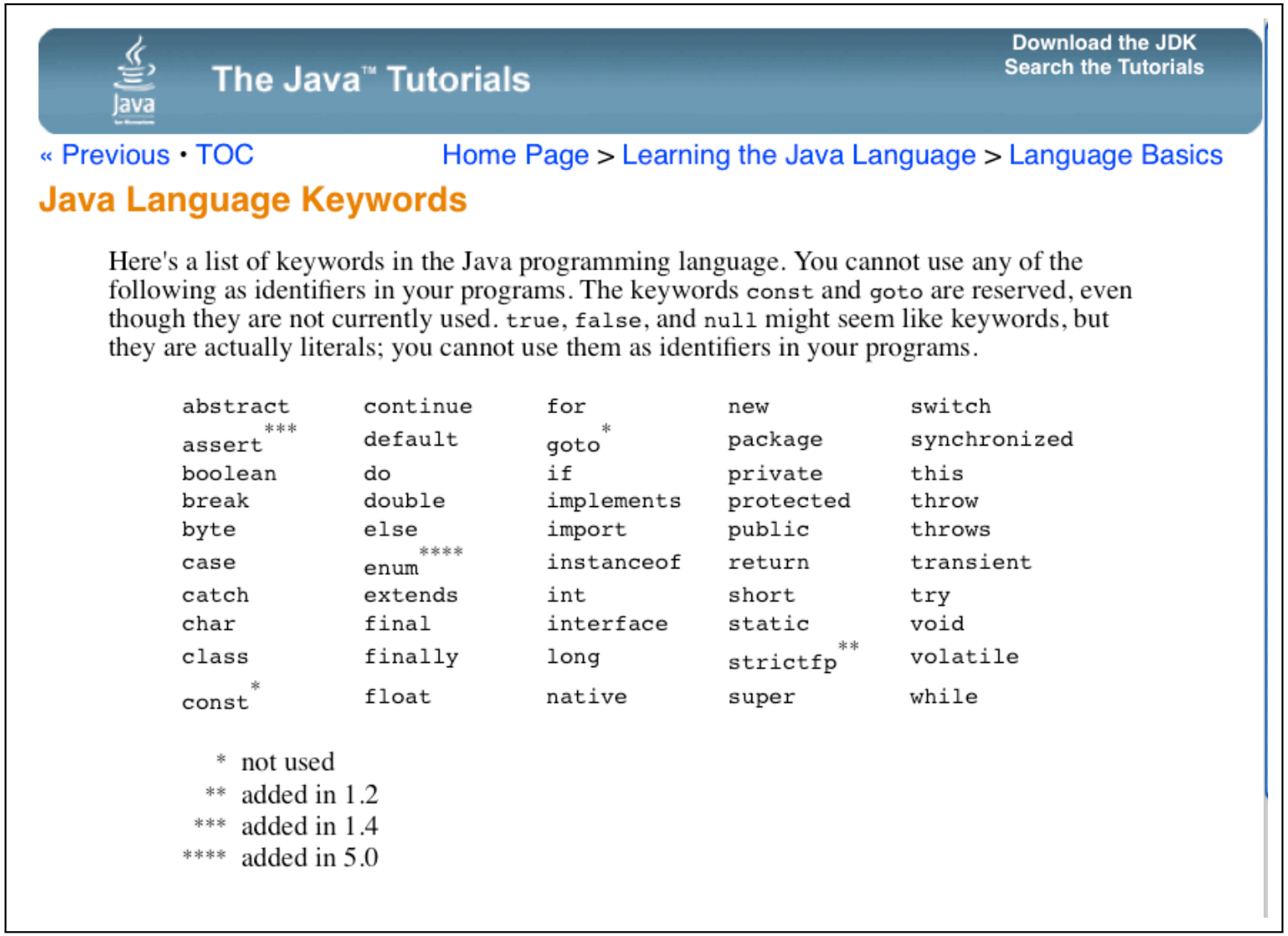

Allmost all computer languages have reserved identifiers, which can not be used as regular identifiers.

```
static = 99; // NOT VALID
```

KU

# Java Keywords

From the online Java tuturials



The Java™ Tutorials

Download the JDK
Search the Tutorials

« Previous • TOC  Home Page > Learning the Java Language > Language Basics

## Java Language Keywords

Here's a list of keywords in the Java programming language. You cannot use any of the following as identifiers in your programs. The keywords `const` and `goto` are reserved, even though they are not currently used. `true`, `false`, and `null` might seem like keywords, but they are actually literals; you cannot use them as identifiers in your programs.

| | | | | |
|---|---|---|---|---|
| abstract | continue | for | new | switch |
| assert*** | default | goto* | package | synchronized |
| boolean | do | if | private | this |
| break | double | implements | protected | throw |
| byte | else | import | public | throws |
| case | enum**** | instanceof | return | transient |
| catch | extends | int | short | try |
| char | final | interface | static | void |
| class | finally | long | strictfp** | volatile |
| const* | float | native | super | while |

* not used
** added in 1.2
*** added in 1.4
**** added in 5.0

# Shortcomings of our grammar

- Only handles a single method
- Only handles methods with one argument
- . . . with type `String[]`
- The function call has to have three names
- The function has one argument
- . . . that has to be a single literal string

```
java ::= public class name { method }
method ::= public static void name ( method_arg ) { statement ; }
method_arg = type name
type ::= String [ ]
statement ::= full_name ( fun_arg )
full_name ::= name . name . name
fun_args ::= string
```

# Shortcomings of our grammar

- Only handles a single method – want any number of methods
- Only handles methods with one argument – want any number of arguments
- . . . with type `String[]`
- The function call has to have three names – want one or more names here
- The function has one argument – want arbitrary number of expressions
- . . . that has to be a single literal string

```
java ::= public class name { method }
method ::= public static void name ( method_arg ) { statement ; }
method_arg = type name
type ::= String [ ]
statement ::= full_name ( fun_arg )
full_name ::= name . name . name
fun_args ::= string
```

# Using BNF to Represent Sequences

$$number_1, number_2, \ldots, number_n$$

- How do we represent one or more numbers ($n >= 1$)?
- How do we represent zero or more numbers ($n >= 0$)?

# Using BNF to Represent Sequences (with 1 or more elements)

```
num_seq ::=      number
```

# Using BNF to Represent Sequences (with 1 or more elements)

```
num_seq ::=       number
          | number , number
```

# Using BNF to Represent Sequences (with 1 or more elements)

```
num_seq ::=       number
          | number , number
          | number , number , number
```

# Using BNF to Represent Sequences (with 1 or more elements)

```
num_seq ::=       number
          | number , number
          | number , number , number
          | ...
```

# Using BNF to Represent Sequences (with 1 or more elements)

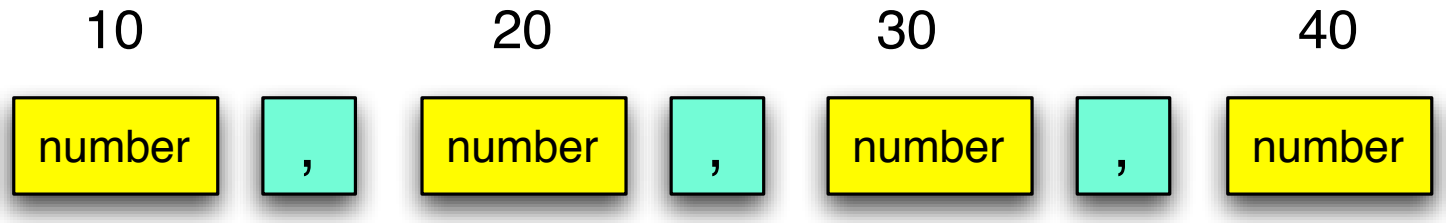
Three styles of non–empty sequences using BNF

```
num_seq ::= number
          | num_seq , num_seq
```

```
num_seq ::= number
          | number , num_seq
```

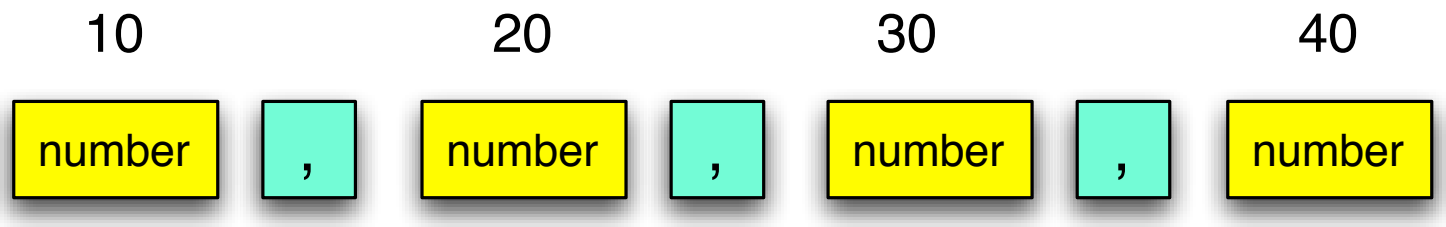
```
num_seq ::= number
          | num_seq , number
```

# Ambiguous Grammar




```
num_seq ::= number
          | num_seq , num_seq
```

# Unambiguous Grammar

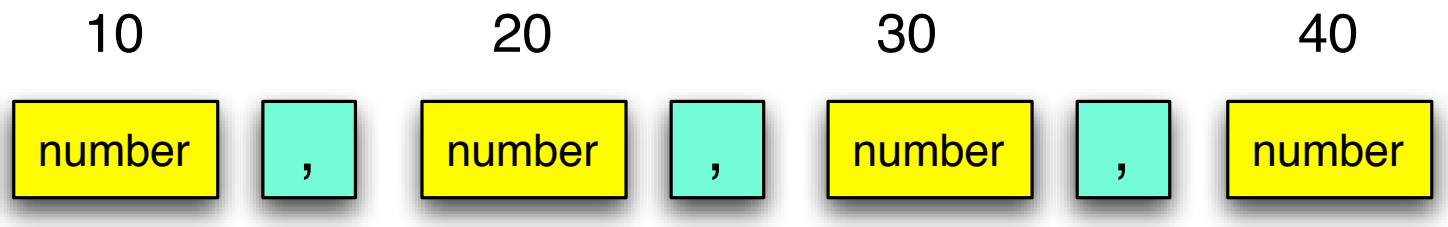



Only one valid syntax tree for any possible input stream

```
num_seq ::= number
          | number , num_seq
```

This grammar produces right leaning trees.

# Unambiguous Grammar




Only one valid syntax tree for any possible input stream

```
num_seq ::= number
          | num_seq , number
```

This grammar produces left leaning trees.

# Summary of non–empty sequences

- Unambiguous grammars produce either left or right leaning trees.
- Ambiguous grammars produce either.

- Real languages need unambiguous specifications!

What about empty-sequences?

KU

# Using BNF to Represent Sequences (with 0 or more elements)

First attempt at three styles of possibly empty sequences using BNF

```
num_seq ::= number
          | num_seq , num_seq
          |
```

```
num_seq ::= number
          | number , num_seq
          |
```

```
num_seq ::= number
          | num_seq , number
          |
```

- All these examples are wrong!

# Using BNF to Represent Sequences (with 0 or more elements)

```
num_seq ::= number
          | num_seq , num_seq
          |
```

- The first grammar accepts , , ,
- Not what we want – a comma separated list

# Using BNF to Represent Sequences (with 0 or more elements)

```
num_seq ::= number
          | number , num_seq
          |
```

- The second grammar accepts 9,10,
- Not what we want – a comma seperated list

# Using BNF to Represent Sequences (with 0 or more elements)

```
num_seq ::= number
          | num_seq , number
          |
```

- The third grammar accepts ,9,10
- Again, not what we want – a comma separated list

# Using BNF to Represent Sequences (with 0 or more elements)

Solution:

- A non–terminal for possibly empty sequences
- A non–terminal for never empty sequences
- Define one in in terms of the other

**Wrong**

```
num_seq ::= number
          | number , num_seq
          |
```

**Revised**

```
num_seq ::= non_empty_num_seq
          |
non_empty_num_seq ::= number
                    | number , non_empty_num_seq
```

# Putting it together

```
java ::= public class name { methods }
```

# Putting it together

```
java ::= public class name { methods }

methods ::= method methods
          |
```

# Putting it together

```
java ::= public class name { methods }

methods ::= method methods
          |

method ::= public static void name ( method_args ) { statement ; }
```

# Putting it together

```
java ::= public class name { methods }

methods ::= method methods
          |

method ::= public static void name ( method_args ) { statement ; }

method_args ::= non_empty_method_args
              |

non_empty_method_args ::= method_arg
                        | method_arg , non_empty_method_args
```

# Putting it together

```
java ::= public class name { methods }

methods ::= method methods
          |

method ::= public static void name ( method_args ) { statement ; }

method_args ::= non_empty_method_args
              |

non_empty_method_args ::= method_arg
                        | method_arg , non_empty_method_args

method_arg ::= ...
```

KU

# Summary

- Started developing a grammar for a real language
- Ambiguous grammars
- Unambiguous grammars
- Representing sequences in BNF

KU