

EECS 368

Programming Language Paradigms

Dr. Andy Gill

Department of Electrical Engineering & Computer Science
University of Kansas

August 31, 2015

Syntax for Languages

C++

```
#include <iostream.h>
main()
{
    cout << "Hello World!";
    return 0;
}
```

Java

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

JavaScript

```
console.log("Hello, World");
```

Scheme

```
(display "Hello World!")
(newline)
```

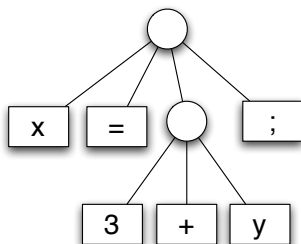
Haskell

```
main :: IO ()
main = putStrLn "Hello World!"
```

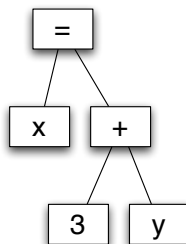
Concrete and Abstract Syntax Trees

Concrete Syntax Trees contains the original textual information

Abstract Syntax Trees have an abstracted, idealized view of a program

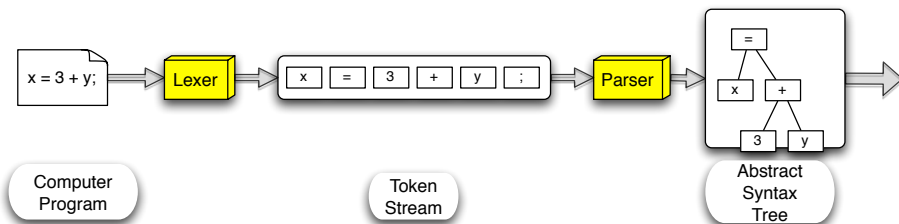


Concrete Syntax Tree

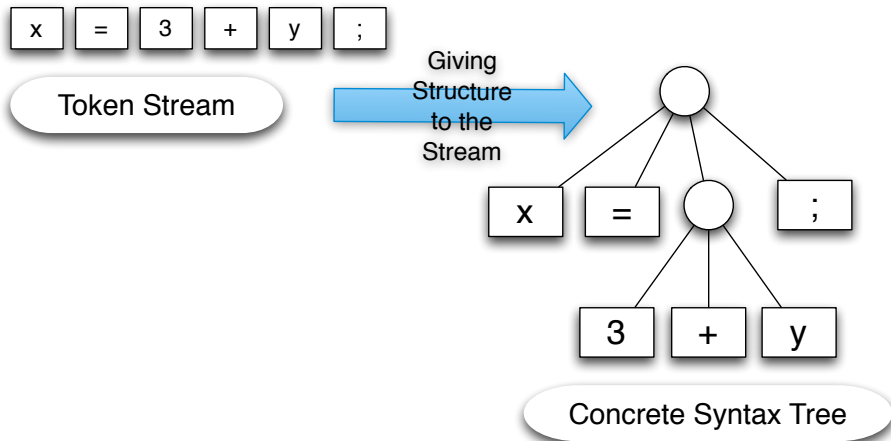


Abstract Syntax Tree

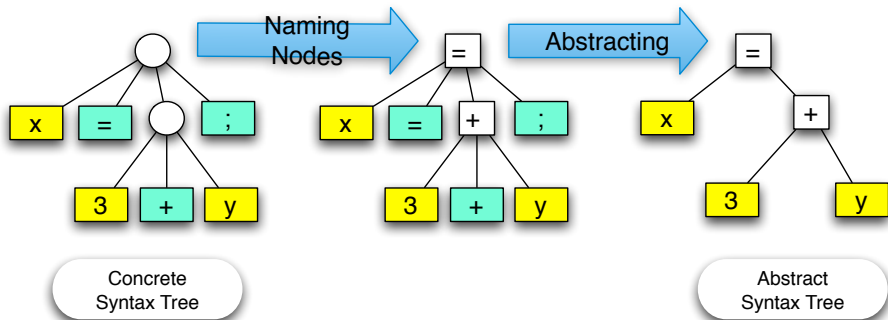
Generating Syntax Trees



Discovering Structure in Tokens



Concrete to Abstract Trees



Inside real compilers, we

- Turn our computer program into a stream of tokens
- Understand the hidden structure of our stream using a notation that expresses valid concrete syntax trees
- When discovering this structure, we build an abstract syntax tree internally

Backus-Naur Form (BNF) is way of expressing valid concrete syntax trees

Backus-Naur Form

- **Non Terminals** are **internal nodes** in a concrete syntax
- S and E are the **Non Terminals**
- **Terminals** are **leaf nodes** in a concrete syntax
- id, =, ;, num, +, *, (and) are the **Terminals**
- id and num are special terminals, with a payload

Our token stream for our example

x = 3 + y;

Actually is a stream of terminals, with optional payloads.



Example BNF

S ::= id = E ;

E ::= id

| num

| E + E

| E * E

| (E)

Discovering Structure



BNF for Statements

$S ::= id = E ;$

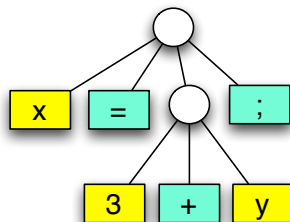
$E ::= id$

| num

| E + E

| E * E

| (E)



Concrete
Syntax Tree

How do we take a token stream, and turn it into a concrete syntax tree?

Production Rules

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

$E \rightarrow \text{id}$

$E \rightarrow \text{num}$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

BNF for Statements

$S ::= \text{id} = E ;$

$E ::= \text{id}$

| num

| $E + E$

| $E * E$

| (E)

Finding Structure

S \rightarrow **id** = **E** ;

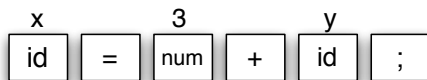
E \rightarrow id

E \rightarrow num

E \rightarrow **E** + **E**

E \rightarrow **E** * **E**

E \rightarrow (**E**)



S

Finding Structure

$S \rightarrow id = E ;$

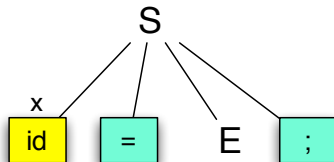
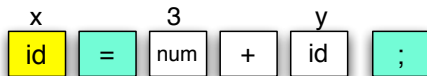
$E \rightarrow id$

$E \rightarrow num$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$



Finding Structure

$S \rightarrow id = E ;$

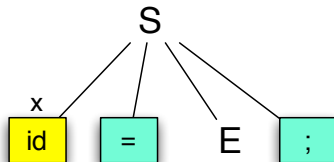
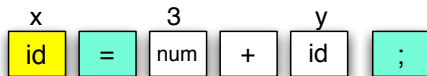
$E \rightarrow id$

$E \rightarrow num$

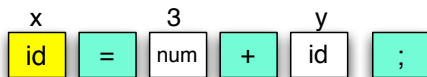
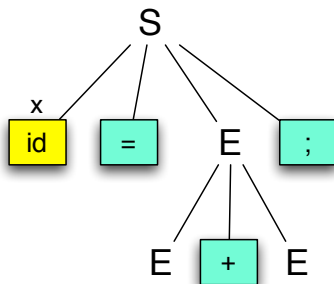
$E \rightarrow E + E$

$E \rightarrow E * E$

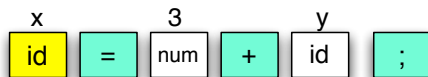
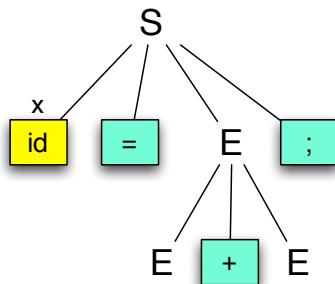
$E \rightarrow (E)$



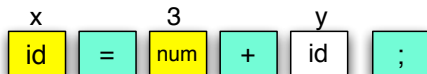
Finding Structure


$$S \rightarrow \text{id} = E ;$$
$$E \rightarrow \text{id}$$
$$E \rightarrow \text{num}$$
$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow (E)$$


Finding Structure


$$S \rightarrow \text{id} = E ;$$
$$E \rightarrow \text{id}$$
$$E \rightarrow \text{num}$$
$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow (E)$$


Finding Structure



$S \rightarrow id = E ;$

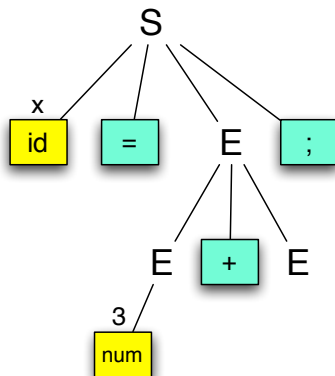
$E \rightarrow id$

$E \rightarrow num$

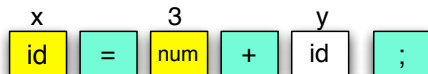
$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$



Finding Structure



$S \rightarrow id = E ;$

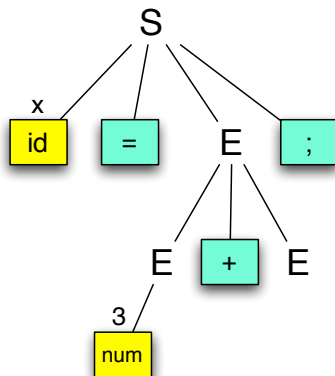
$E \rightarrow id$

$E \rightarrow num$

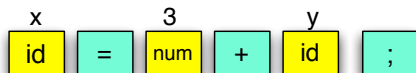
$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$



Finding Structure



$S \rightarrow id = E ;$

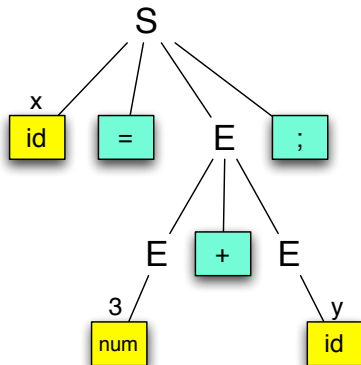
$E \rightarrow id$

$E \rightarrow num$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$



Summary

- Syntax Trees are used to represent programs
- BNF is way of expressing possible concrete syntax trees
- BNF can be used to guide turning a token stream into a concrete syntax tree

Exercise: Use our BNF example to find the concrete syntax tree

`x = (2 * (3 * y)) ;`

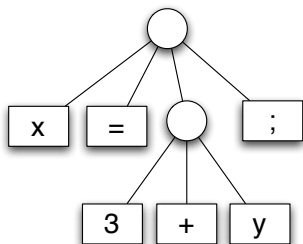
Example BNF

```
S ::= id = E ;  
E ::= id  
    |  num  
    |  E + E  
    |  E * E  
    |  ( E )
```

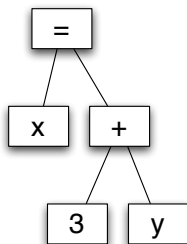
A Syntax Tree

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

$x = 3 + y;$

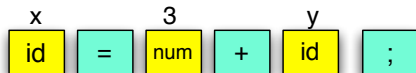


Concrete Syntax Tree



Abstract Syntax Tree

Finding Structure



$S \rightarrow id = E ;$

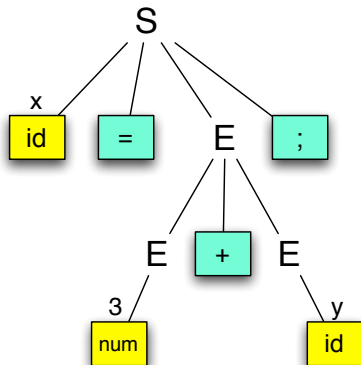
$E \rightarrow id$

$E \rightarrow num$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$



**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

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!");  
    }  
}
```

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 []`

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 ; }`

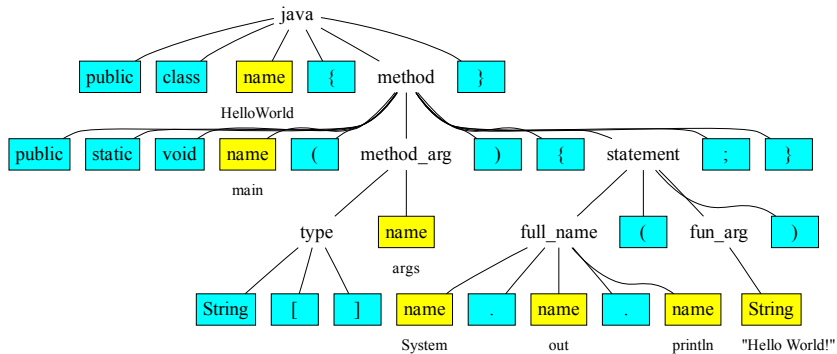
`method_arg ::= type name`

`type ::= String []`

`statement ::= full_name (fun_arg)`

`full_name ::= name . name . name`

`fun_arg ::= string`



Keywords

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

HelloWorld



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

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




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.

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

* 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
```

Recap: Syntax for Languages

C++

```
#include <iostream.h>
main()
{
    cout << "Hello World!";
    return 0;
}
```

Java

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

JavaScript

```
console.log("Hello, World");
```

Scheme

```
(display "Hello World!")
(newline)
```

Haskell

```
main :: IO ()
main = putStrLn "Hello World!"
```

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

number₁, number₂, . . . , number_n

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

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?

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 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
```