

# Announcements

Working in pairs is only allowed for programming assignments and not for homework problems

H3 has been posted

# Syntax Directed Translation

# CFGs so Far

## CFGs for Language *Definition*

- The CFGs we've discussed can generate/define languages of valid strings
- So far, we **start** by building a parse tree and **end** with some valid string

## CFGs for Language *Recognition*

- Start with a string and end with a parse tree for it

# CFGs for Parsing

Language Recognition isn't enough for a parser

- We also want to *translate* the sequence

Parsing is a special case of *Syntax-Directed Translation*

- Translate a sequence of tokens into a sequence of actions

Syntax-directed translation (SDT) takes a parse tree and output something else. This can be string, a integer value, etc. When the output is an abstract-syntax tree, this process is parsing.

The abstract-syntax tree is the output of parsing, used in the next phase of compiling.

# Syntax Directed Translation

Augment CFG rules with translation rules (at least 1 per production)

A translation rule Define translation of LHS nonterminal as function of

- Constants
- RHS nonterminal translations
- RHS terminal value

Assign rules bottom-up

To translate a input string into an abstract-syntax tree:  
(1) build the parse tree;  
(2) apply the translation rules to compute the translation value for each non-terminals in the tree, working bottom up (since a nonterminal's value may depend on the value of the symbols on the right-hand side, you need to work bottom-up so that those values are available).

# SDT Example

".trans" means translation.

CFG

$B \rightarrow 0$

| 1

|  $B0$

|  $B1$

Rules

$B.trans = 0$

$B.trans = 1$

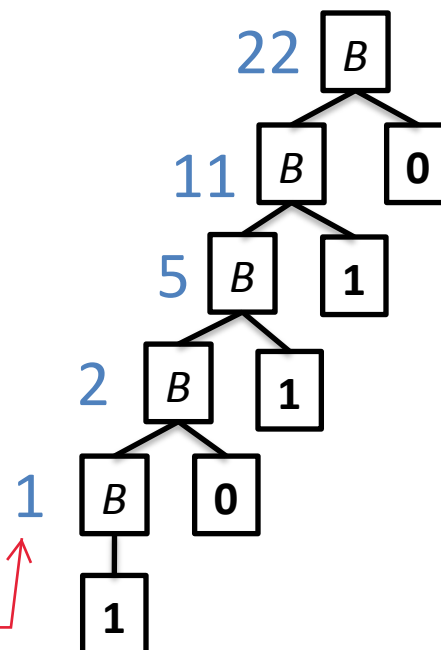
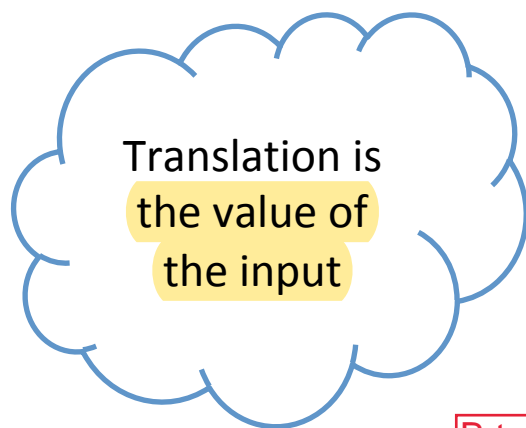
$B.trans = B_2.trans * 2$

$B.trans = B_2.trans * 2 + 1$

Input string

10110

Assume that we already have the parse tree.



# SDT Example 2: Declarations

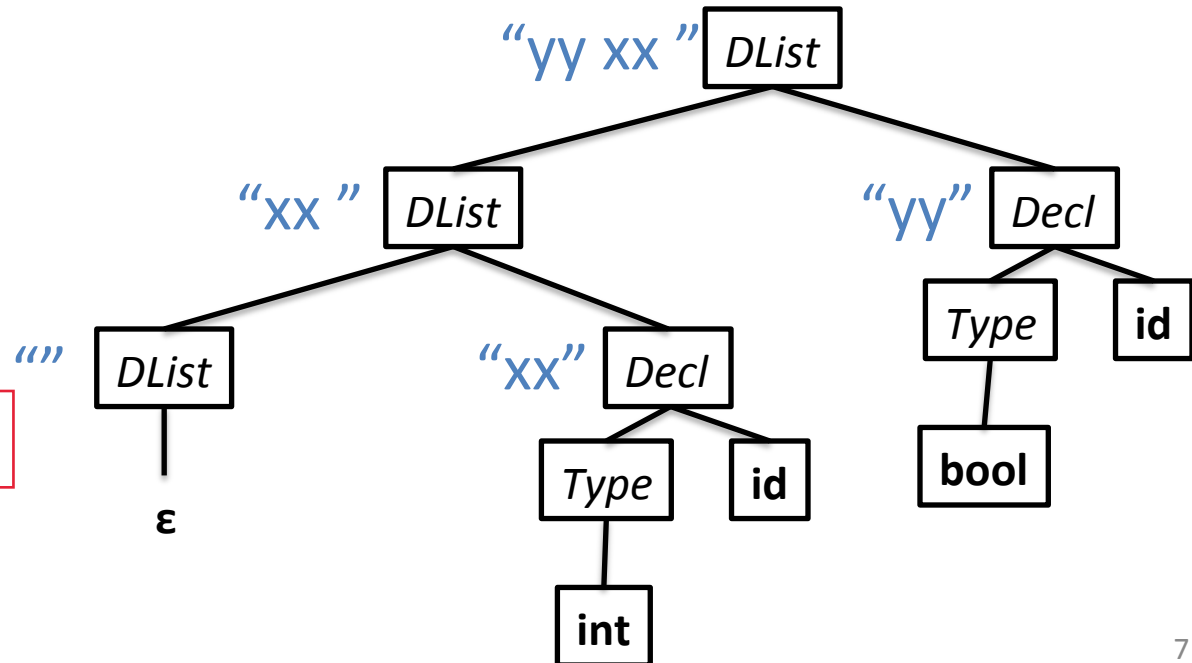
Translation is a  
String of ids

CFG

$DList$	$\rightarrow \epsilon$	$DList.trans = ""$
	$  DList Decl$	$DList.trans = Decl.trans + " " + DList_2.trans$
$Decl$	$\rightarrow Type id$	$Decl.trans = id.value$
$Type$	$\rightarrow int$	
	$  bool$	

Input string

int xx;  
bool yy;



Syntax directed translation:  
get something from the parse tree.

# Exercise Time

Only add declarations of type `int` to the output String.

Augment the previous grammar:

## CFG

*DList*     $\rightarrow \epsilon$   
          | *DList Decl*  
*Decl*     $\rightarrow$  *Type id ;*  
*Type*     $\rightarrow$  **int**  
          | **bool**

## Rules

*DList.trans* = ""  
*DList.trans* = *Decl.trans* + " " + *DList*<sub>2</sub>.*trans*  
*Decl.trans* = **id.value**

Different nonterms can  
have different types

Rules can have conditionals



# SDT Example 2b: ints only

Translation is a  
String of **int** ids  
only

## CFG

*DList*  $\rightarrow \epsilon$   
| *Decl DList*  
*Decl*  $\rightarrow$  *Type id* ;  
*Type*  $\rightarrow$  *int*  
| *bool*

## Rules

*DList.trans* = ""

*DList.trans* = *Decl.trans* + " " + *DList<sub>2</sub>.trans*

if (*Type.trans*) { *Decl.trans* = *id.value* } else { *Decl.trans* = "" }

*Type.trans* = true

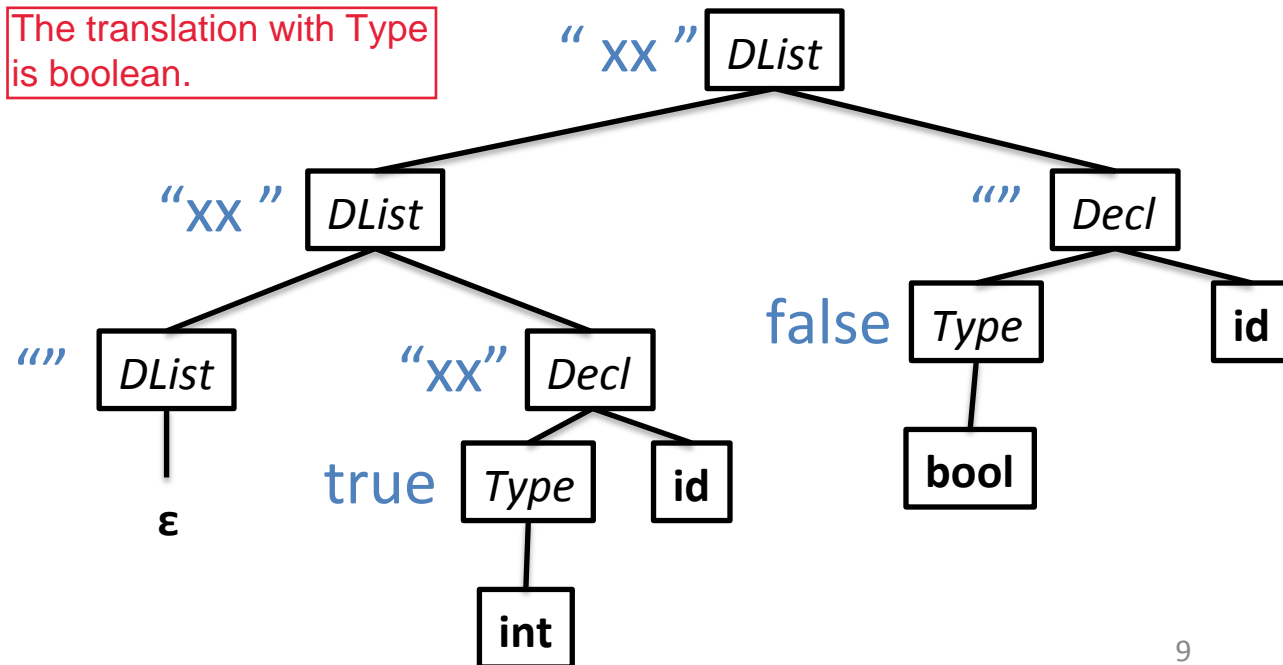
*Type.trans* = false

Why not return the string directly? It is ok though,  
but strings are expensive. Booleans are better :).

The translation with *Type*  
is boolean.

## Input string

int xx;  
bool yy;



Different nonterms can  
have different types

Rules can have conditionals

# SDT for Parsing

In the previous examples, the SDT process assigned different types to the translation:

- Example 1: tokenized stream to an **integer value**
- Example 2: tokenized stream to a (java) **String**

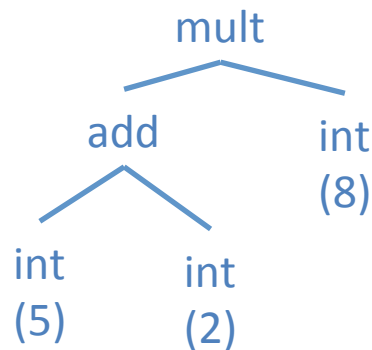
For parsing, we'll go from tokens to an Abstract-Syntax Tree (AST)

Goal:  
parse tree -> abstract  
syntax tree.

# Abstract Syntax Trees

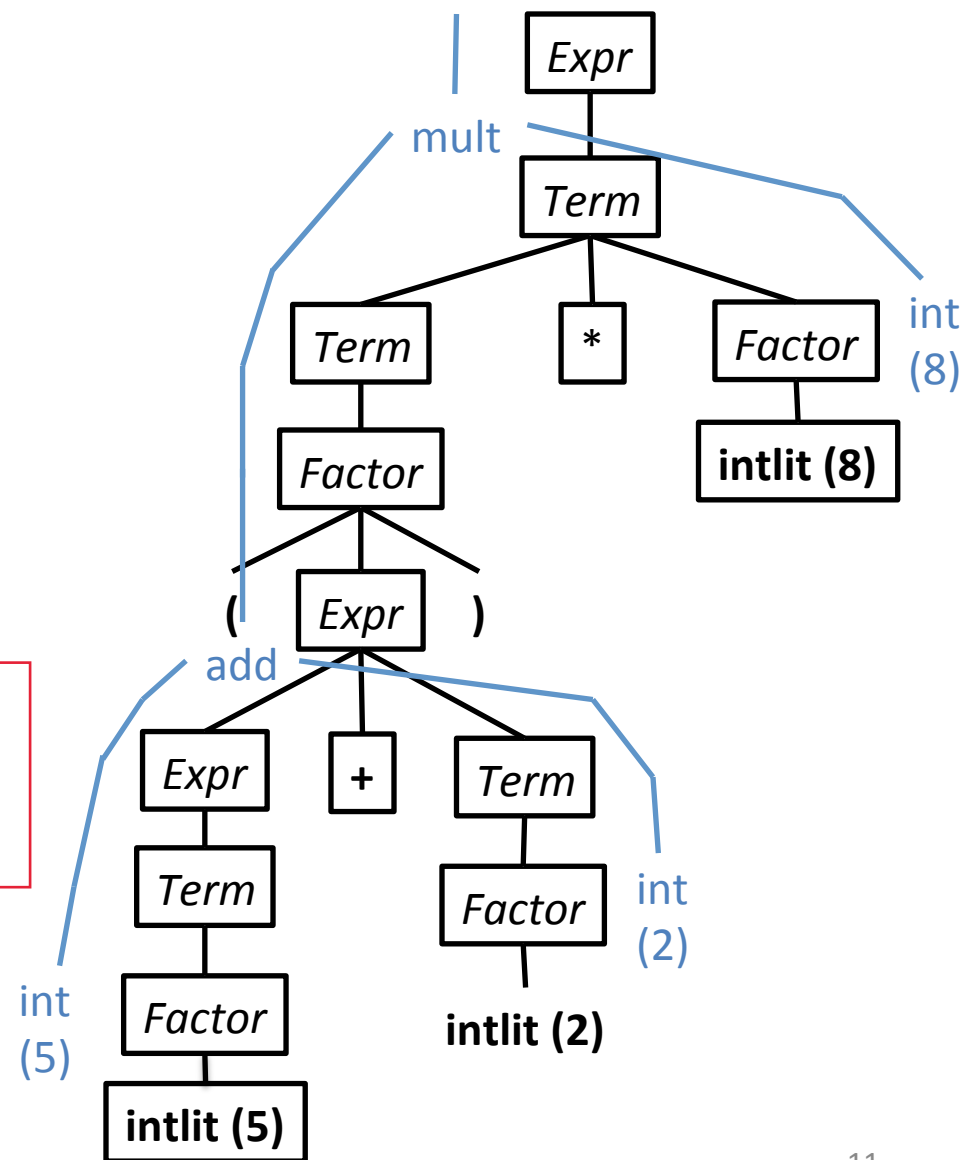
- A condensed form of the parse tree
- Operators at internal nodes (not leaves)
- Chains of productions are collapsed
- Syntactic details omitted

Example:  $(5+2)*8$



Should the translation do type checking?  
No. Not in this phase.

Parse Tree



# Exercise #2

- Show the AST for:  
 $(1 + 2) * (3 + 4) * 5 + 6$

Expr  $\rightarrow$  Expr + Term  
| Term

Term  $\rightarrow$  Term \* Factor  
| Factor

Factor  $\rightarrow$  intlit MkIntNode(intlit.value)  
| ( Expr )

Expr  $\rightarrow$  Expr + Term      *Expr1*.trans = MkPlusNode(*Expr2*.trans, *Term*.trans)

# AST for Parsing

In previous slides we did our translation in two steps

- Structure the stream of tokens into a parse tree
- Use the parse tree to build an abstract syntax tree, throw away the parse tree

In practice, we will combine these into 1 step

**Question:** Why do we even need an AST?

- More of a “logical” view of the program
- Generally easier to work with than the parse tree.

# AST Implementation

How do we actually represent an AST in code?

# ASTs in Code

Note that we've assumed a field-like structure in our SDT actions:

$$DList.trans = Decl.trans + " " + DList_2.trans$$

In our parser, we'll define classes for each type of nonterminal, and create a new nonterminal in each rule.

- In the above rule we might represent DList as

```
public class DList{  
    public String trans;  
}
```

- For ASTs: when we execute an SDT rule
  - we construct a new node object for the RHS
  - propagate its fields with the fields of the LHS nodes

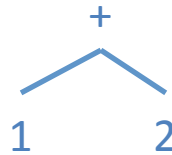
# Thinking about implementing ASTs

Consider the AST for a simple language of Expressions

Input  
1 + 2

Tokenization  
intlrit plus intlrit

AST



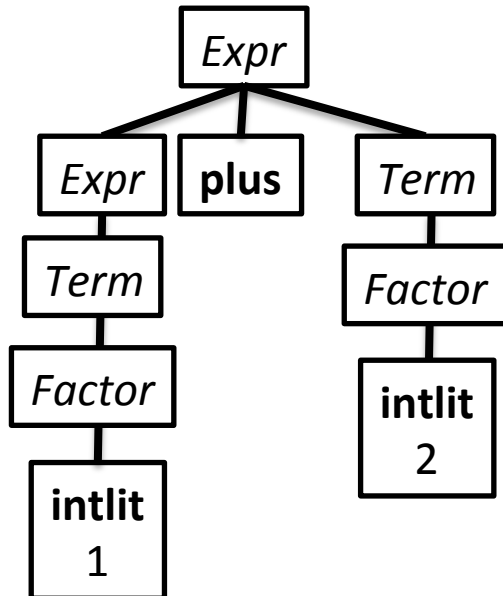
Naïve AST Implementation

```
class PlusNode
    IntNode left;
    IntNode right;
}
```

You cannot have 1+ 2 + 3.

```
class IntNode{
    int value;
}
```

Parse Tree



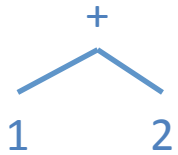


# Thinking about implementing ASTs

Consider AST node classes

- We'd like the classes to have a common inheritance tree

AST

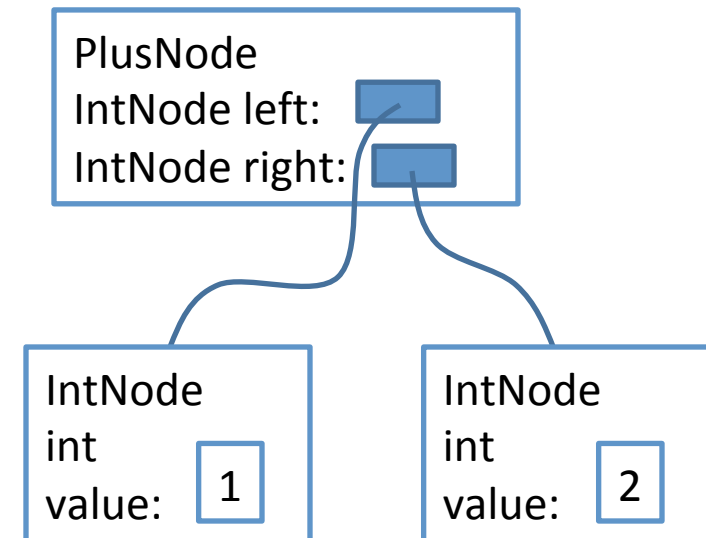


Naïve AST Implementation

```
class PlusNode
{
    IntNode left;
    IntNode right;
}
```

```
class IntNode
{
    int value;
}
```

Naïve java AST

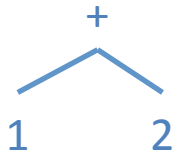


# Thinking about implementing ASTs

Consider AST node classes

- We'd like the classes to have a common inheritance tree

AST



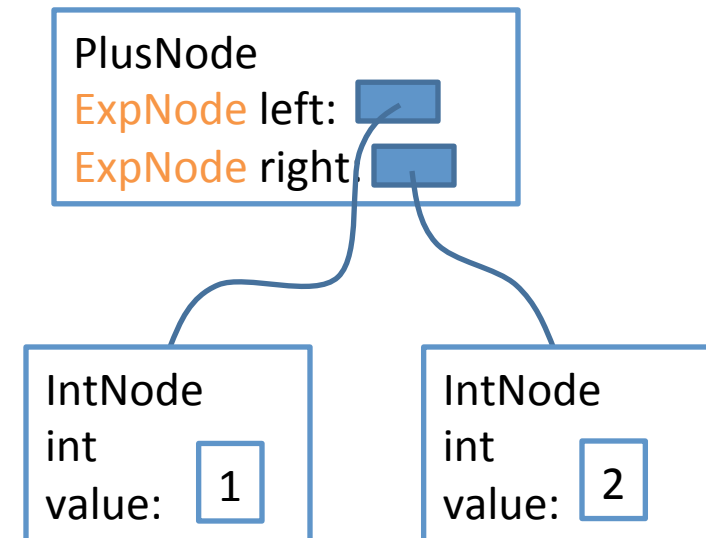
Naïve AST Implementation

```
class PlusNode
{
    IntNode left;
    IntNode right;
}
```

```
class IntNode
{
    int value;
}
```

Make these extend  
ExpNode

Better java AST



# Implementing ASTs for Expressions

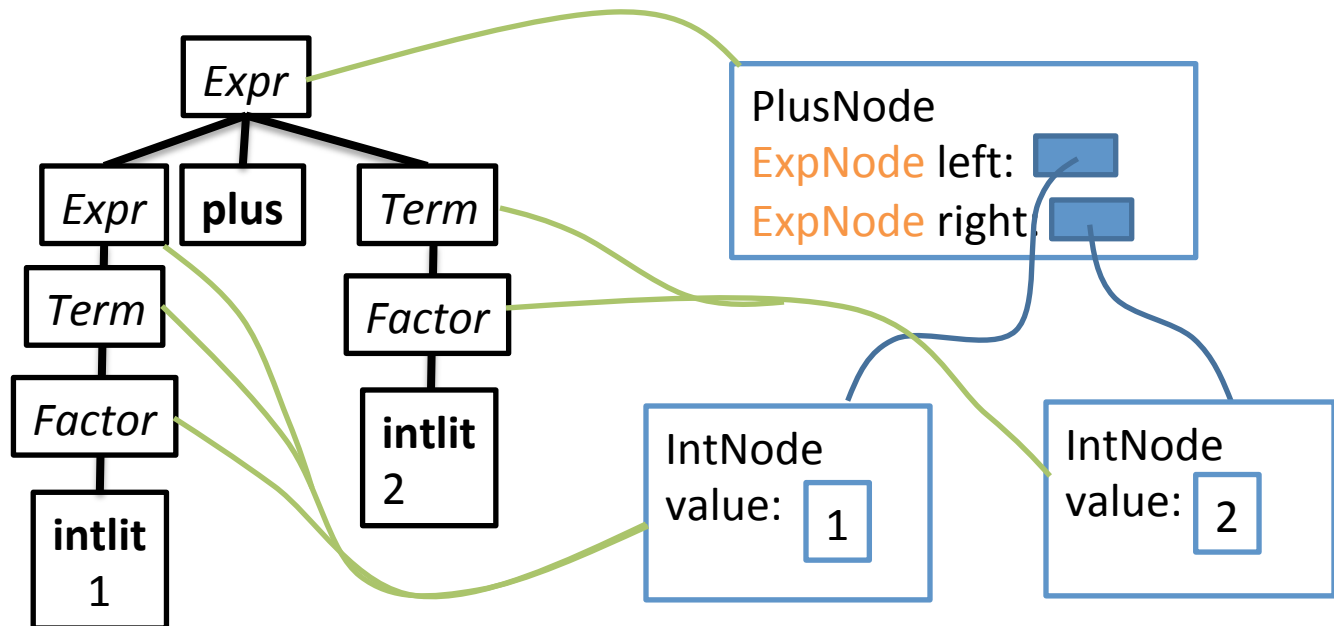
## CFG

Expr  $\rightarrow$  Expr + Term  
| Term  
Term  $\rightarrow$  Term \* Factor  
| Factor  
Factor  $\rightarrow$  intlit  
| ( Expr )

## Translation Rules

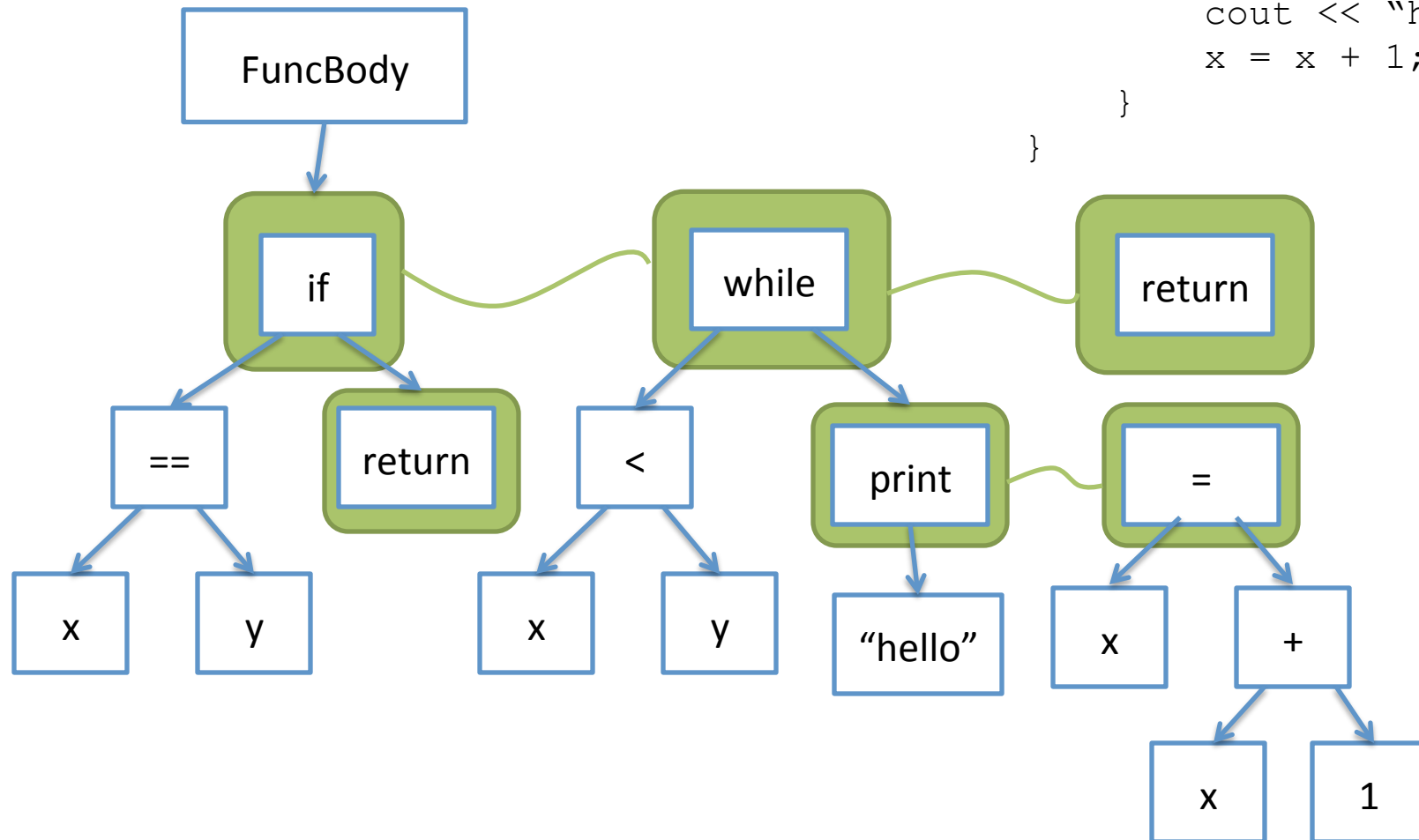
*Expr1.trans* = new PlusNode(*Expr2.trans*, *Term.trans*)  
*Expr.trans* = *Term.trans*  
*Term1.trans* = new TimesNode(*Term2.trans*, *Factor.trans*)  
*Term.trans* = *Factor.trans*  
*Factor.trans* = new IntNode(**intlit.value**)  
*Factor.trans* = *Expr.trans*

Example: 1 + 2



# An AST for a code snippet

```
void foo(int x, int y){  
    if (x == y){  
        return;  
    }  
    while ( x < y){  
        cout << "hello";  
        x = x + 1;  
    }  
}
```



# Summary (1 of 2)

Today we learned about

- Syntax-Directed Translation (SDT)
  - Consumes a parse tree with actions
  - Actions yield some result
- Abstract Syntax Trees (ASTs)
  - The result of SDT for parsing in a compiler
  - Some practical examples of ASTs

# Summary (2 of 2)

## Scanner

Language abstraction: RegEx

Output: Token Stream

Tool: JLex

Implementation: DFA walking via table

## Parser

Language abstraction: CFG

Output: AST by way of Parse Tree

Tool: Java CUP

Implementation: ???

Next time

Build a tree from a string  
and a grammar.

Next week