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 w, and end with ye[illegible]o depending on whether $w \in L(G)$

CFGs in a compiler

- Start with a string w, and end with a parse tree for w if $w \in L(G)$

Generally an abstract-syntax tree rather than a parse tree

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)

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

- Define translation of LHS nonterminal as function of
  - Constants
  - RHS nonterminal translations
  - RHS terminal value

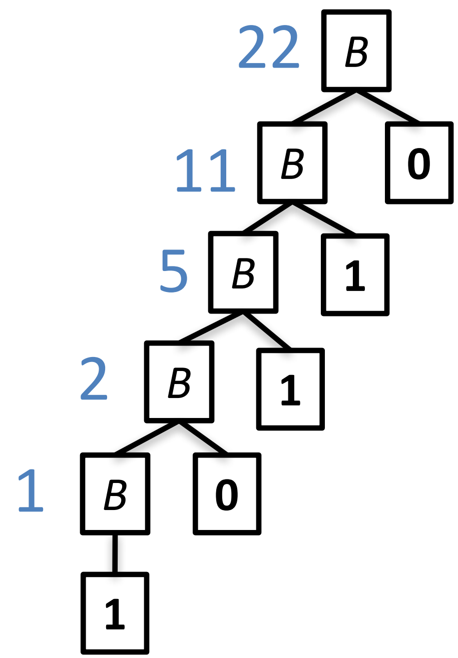
Assign rules bottom-up

SDT Example

CFG	Rules
B -> **0**	B.trans = 0
	1	B.trans = 1
	B **0**	B.trans = B_2.trans * 2
	B **1**	B.trans = B_2.trans * 2 + 1

Input string

10110



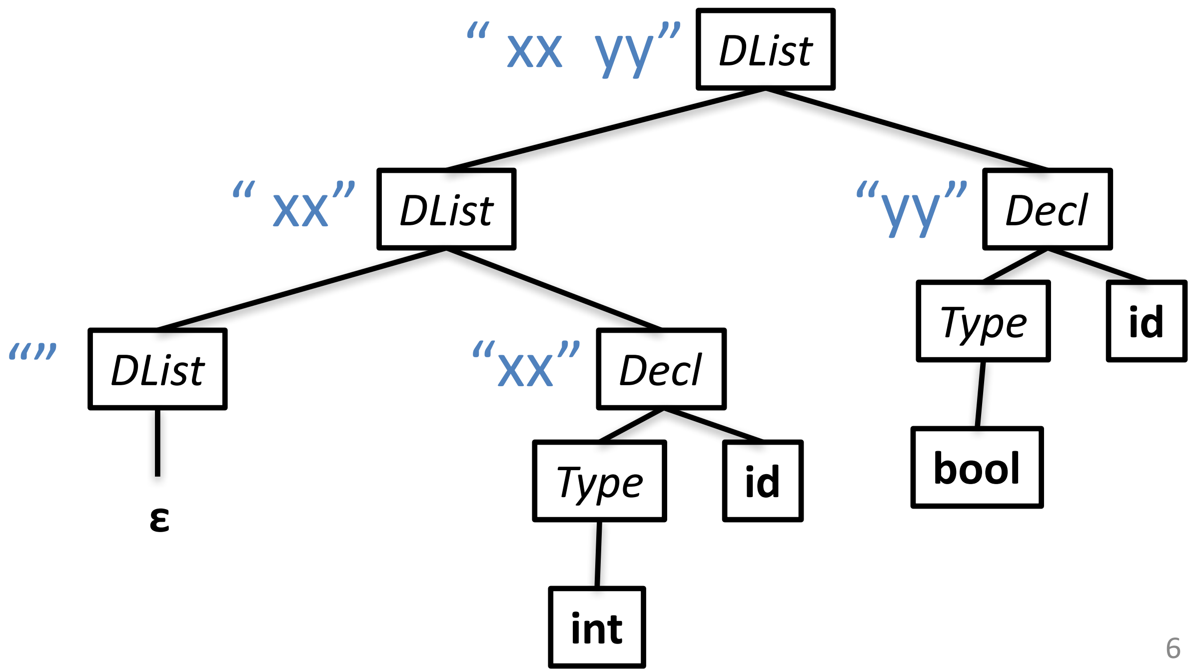
Translation is the value of the input

SDT Example 2: Declarations

Translation is a String of ids

CFG			Rules
DList	→	ε	*DList*.trans = “”
	|	*DList Decl*	*DList.trans* = $DList_2$.trans + “ “ + *Decl.trans*
Decl	→	*Type* **id** ;	*Decl*.trans = **id**.value
Type	→	int	
	|	bool	

Input string
int xx;
bool yy;



Exercise Time

Only add declarations of type int to the output String.

Augment the previous grammar:

CFG			Rules
<u>CFG</u>			<u>Rules</u>
DList	→	**ε**	*DList*.trans = “”
	|	*Decl DList*	*DList.trans* = DList_2.trans + “ “ + *Decl.trans*
Decl	→	*Type* **id ;**	*Decl*.trans = **id**.value
Type	→	**int**	
	|	**bool**	

Different nonterms can have different types

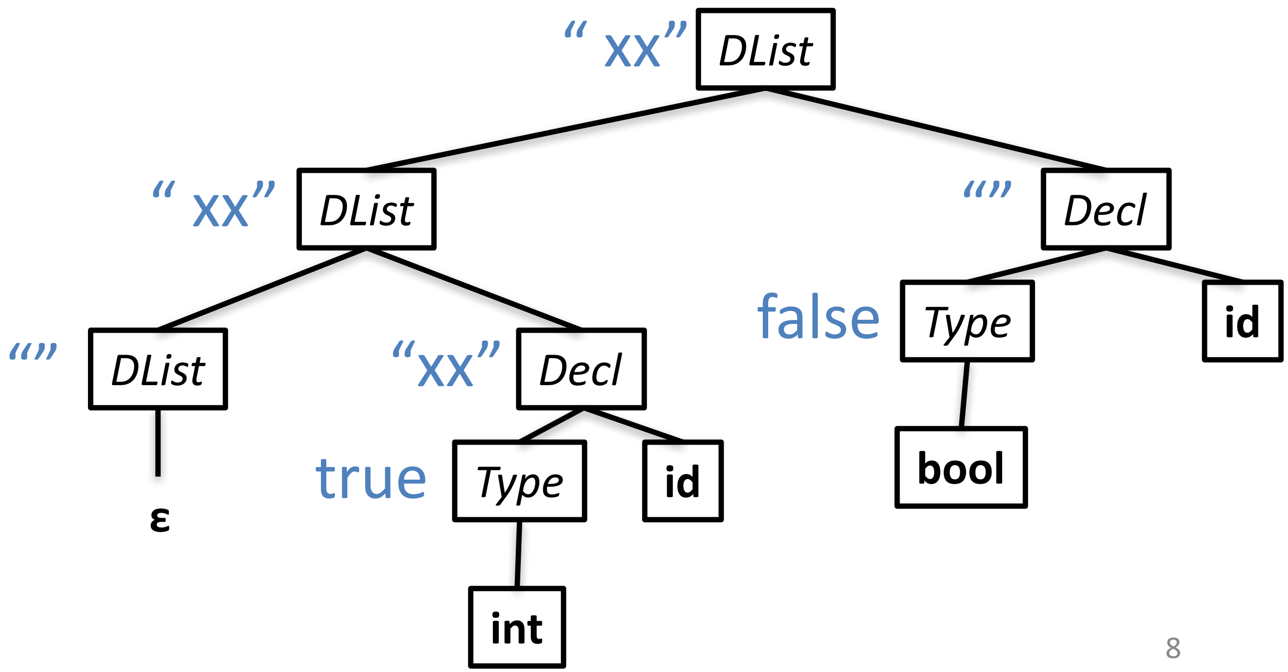
Rules can have conditionals

SDT Example 2b: ints only

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

CFG			Rules
DList	→	**ε**	*DList*.trans = ""
	|	*Decl DList*	*DList.trans* = $DList_2$.trans + " " + *Decl.trans*
Decl	→	*Type* **id ;**	*Decl*.trans = (*Type*.trans ? **id**.value : "")
Type	→	**int**	*Type*.trans = true
	|	**bool**	*Type*.trans = false

Input string
int xx;
bool yy;



Different nonterms can have different types

Rules can use conditional expressions

SDT for Parsing

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

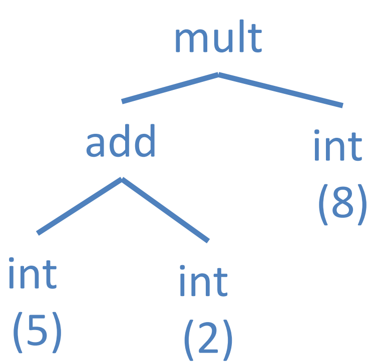
- Example 1: tokenized stream to an **integer value**
- Example 2: tokenized stream to a (Java) **String**

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

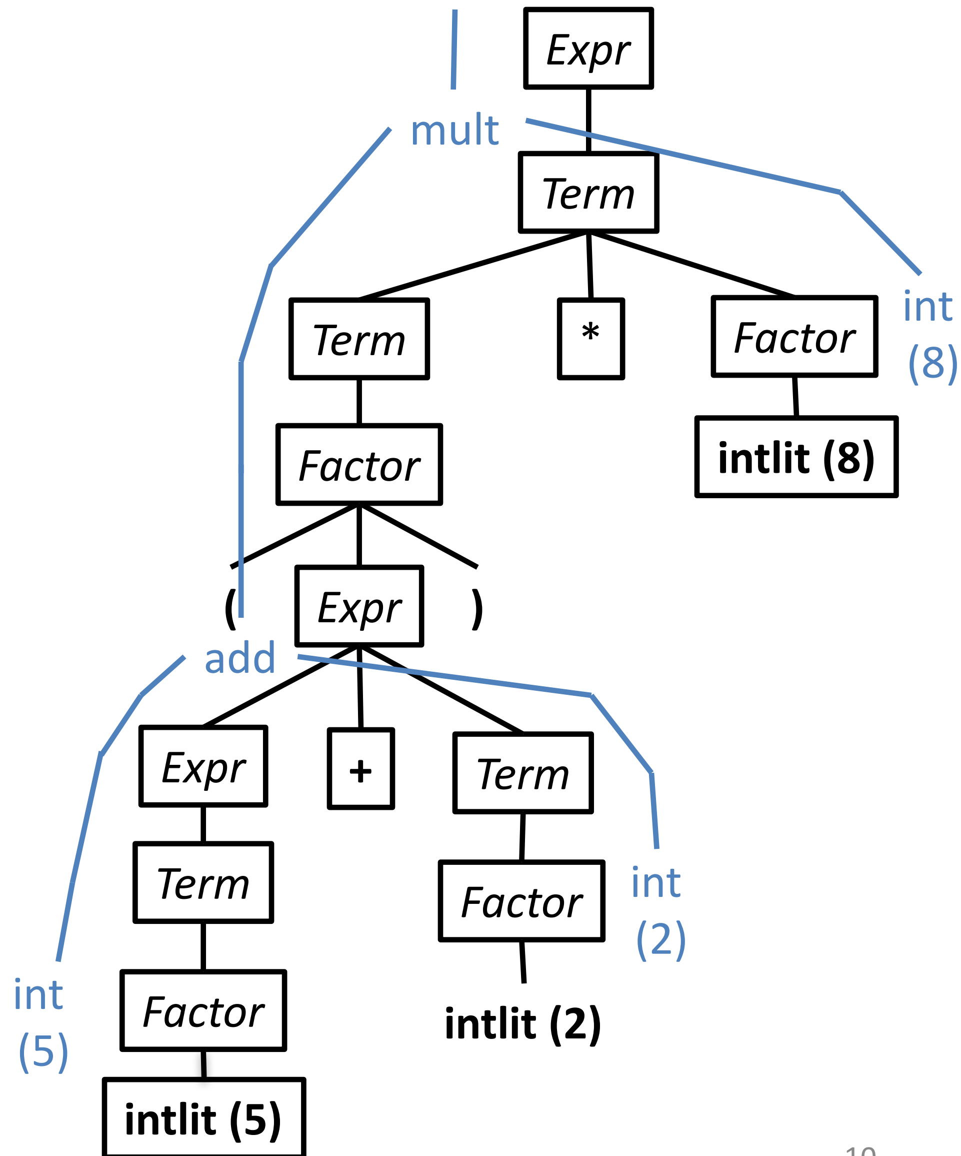
Abstract Syntax Trees

- 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



Parse Tree



Exercise #2

- Show the AST for:

  (1 + 2) * (3 + 4) * 5 + 6

```
Expr   -> Expr + Term
        |  Term
Term   -> Term * Factor
        |  Factor
Factor -> intlit
        |  ( Expr )
```

Expr -> Expr + Term   *Expr1*.trans = MkPlusNode(*Expr2*.trans, *Term*.trans)

AST for Parsing

In previous slides we did the translation in two steps

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

In practice, we will combine these into one step

Question: Why do we even need an AST?

- More of a “logical” view of the program: the essential structure
- Generally easier to work with an AST (in the later phases of name analysis and type checking)
  - no cascades of exp → term → factor → intlit, which was introduced to capture precedence and associativity

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:

Expr -> Expr + Term *Expr1*.trans = MkPlusNode(*Expr2*.trans, *Term*.trans)

In our parser, we'll define a class for each kind of ADT node, and create a new node object in some rules

- In the above rule we would represent the *Expr1*.trans value via the class

```
public class PlusNode extends ExpNode {
   public ExpNode left;
   public ExpNode right;
}
```

- For ASTs: when we execute an SDT rule
  - we construct a new node object, which becomes the value of LHS.trans
  - populate the node's fields with the translations of the RHS nonterminals

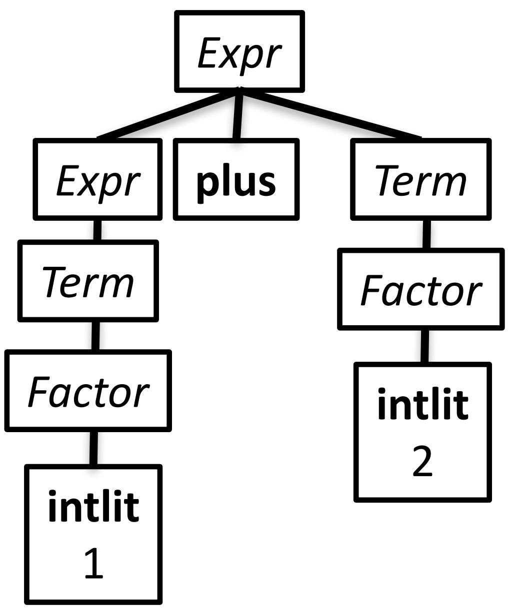
How to implement ASTs

Consider the AST for a simple language of Expressions

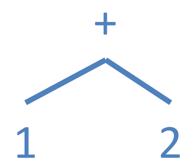
<u>Input</u>
1 + 2

<u>Tokenization</u>
intlit plus intlit

<u>Parse Tree</u>



<u>AST</u>



<u>Naïve AST Implementation</u>

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

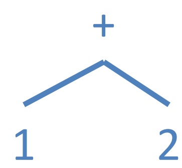
```
class IntNode{
        int value;
}
```

How to implement ASTs

Consider AST node classes

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

AST

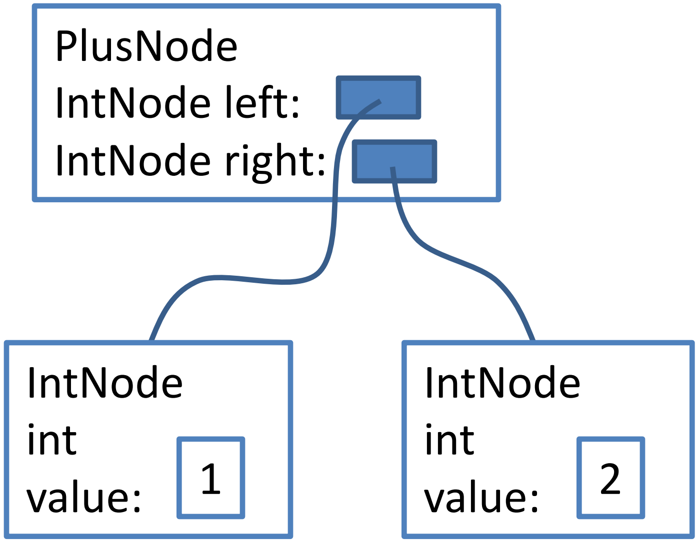


Naïve AST Implementation

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

class IntNode
{       int value;
}
```

Naïve Java AST

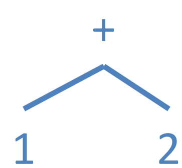


How to implement ASTs

Consider AST node classes

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

<u>AST</u>



<u>Naïve AST Implementation</u>

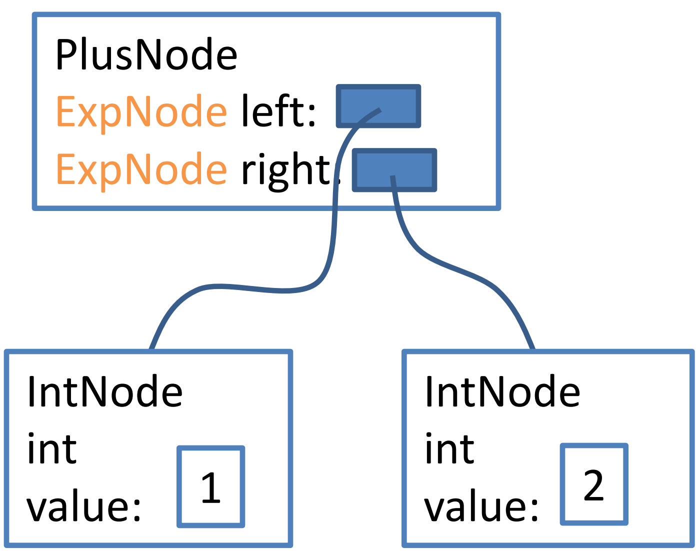
```
class PlusNode
{       IntNode left;
        IntNode right;
}

class IntNode
{       int value;
}
```

Make these extend ExpNode

Make these fields be of class ExpNode

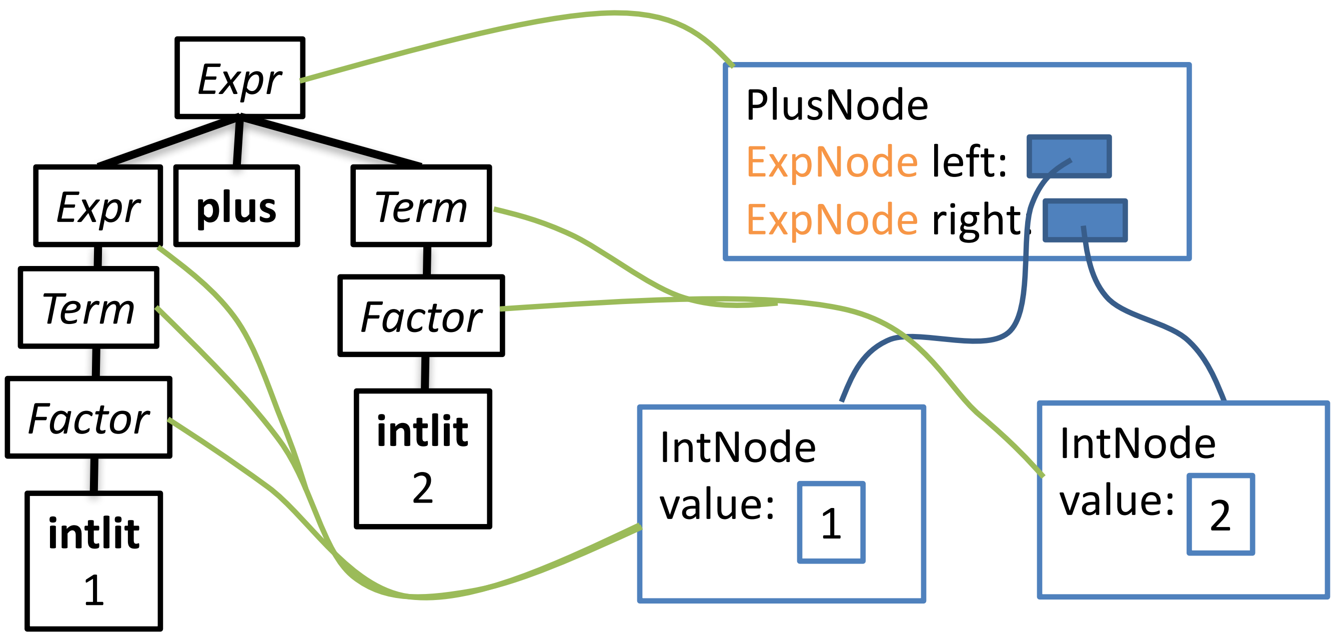
<u>Better Java AST</u>



Implementing ASTs for Expressions

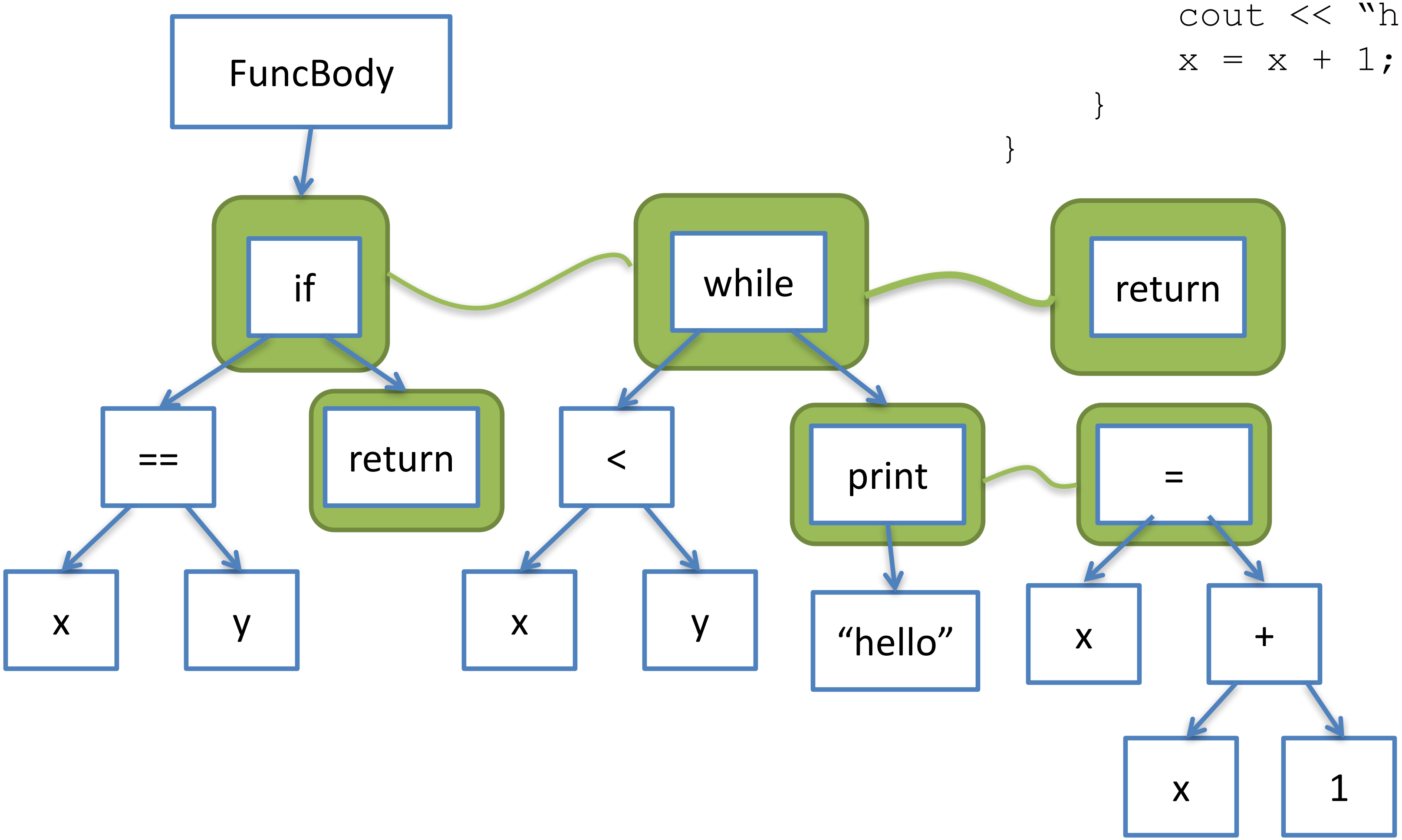
CFG			Translation Rules
Expr	->	Expr + Term	*Expr1*.trans = new PlusNode(*Expr2*.trans, *Term*.trans)
	|	Term	*Expr*.trans = Term.trans
Term	->	Term * Factor	*Term1*.trans = new TimesNode(Term2.trans, *Factor*.trans)
	|	Factor	*Term*.trans = *Factor*.trans
Factor	->	intlit	*Factor*.trans = new IntNode(**intlit**.value)
	|	(Expr)	*Factor*.trans = *Expr*.trans

Example: 1 + 2



An AST for an 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 an SDT performed during parsing in a compiler
  - Some practical examples of ASTs

Summary (2 of 2)

Scanner

Language abstraction: RegExp
Output: Token Stream
Tool: JLex
Implementation: Interpret DFA using table (for δ), recording most_recent_accepted_position and most_recent_token

Parser

Language abstraction: CFG
Output: AST by way of a syntax-directed translation
Tool: Java CUP ← Next week
Implementation: ??? ← Next week