CS536

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

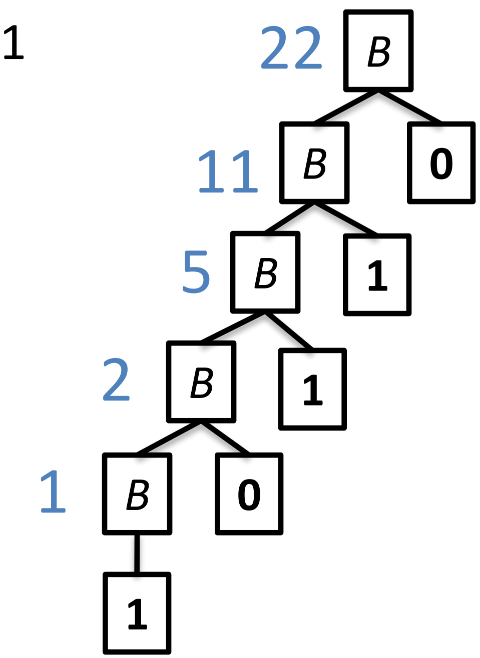
- Augment CFG rules with translation rules (at least 1 per production)
  - Define translation of LHS nonterminal as a function of
    - Constants
    - RHS nonterminal translations
    - RHS terminal value

SDT Example: Binary Numbers

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



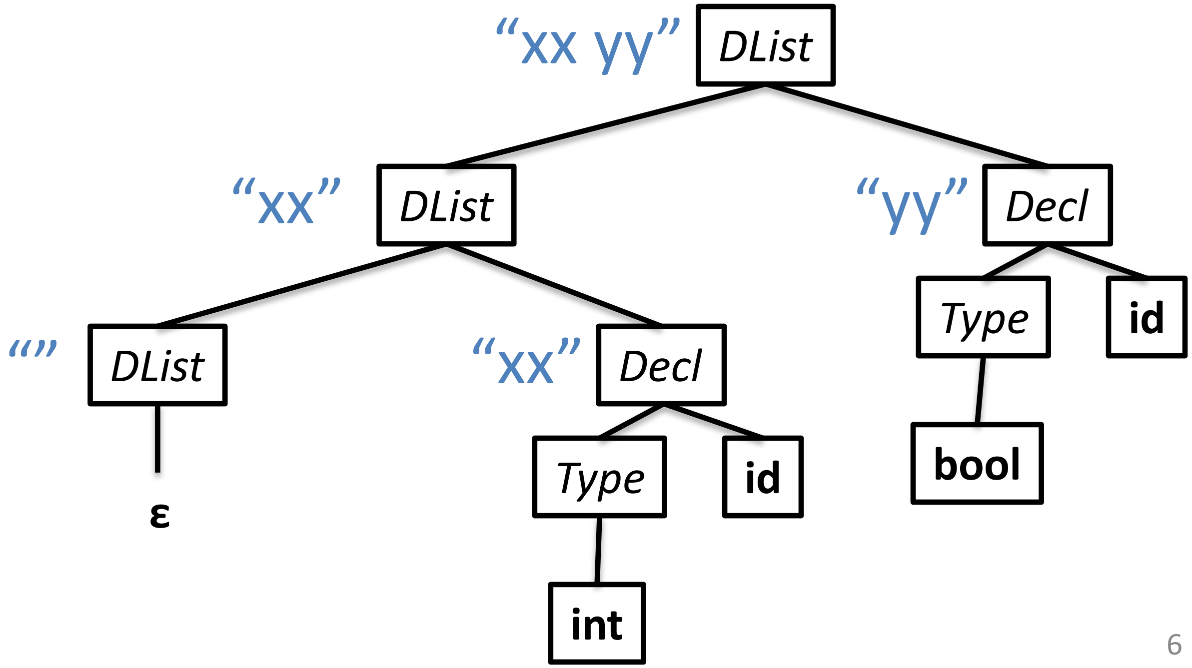
Translation is the value of the input

SDT Example 2: Declarations

Translation is a String of ids

CFG			Rules
DList	→	ε	*DList*.trans = “”
	|	*Decl DList*	*DList.trans* = *Decl.trans* + “ “ + DList_2.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:

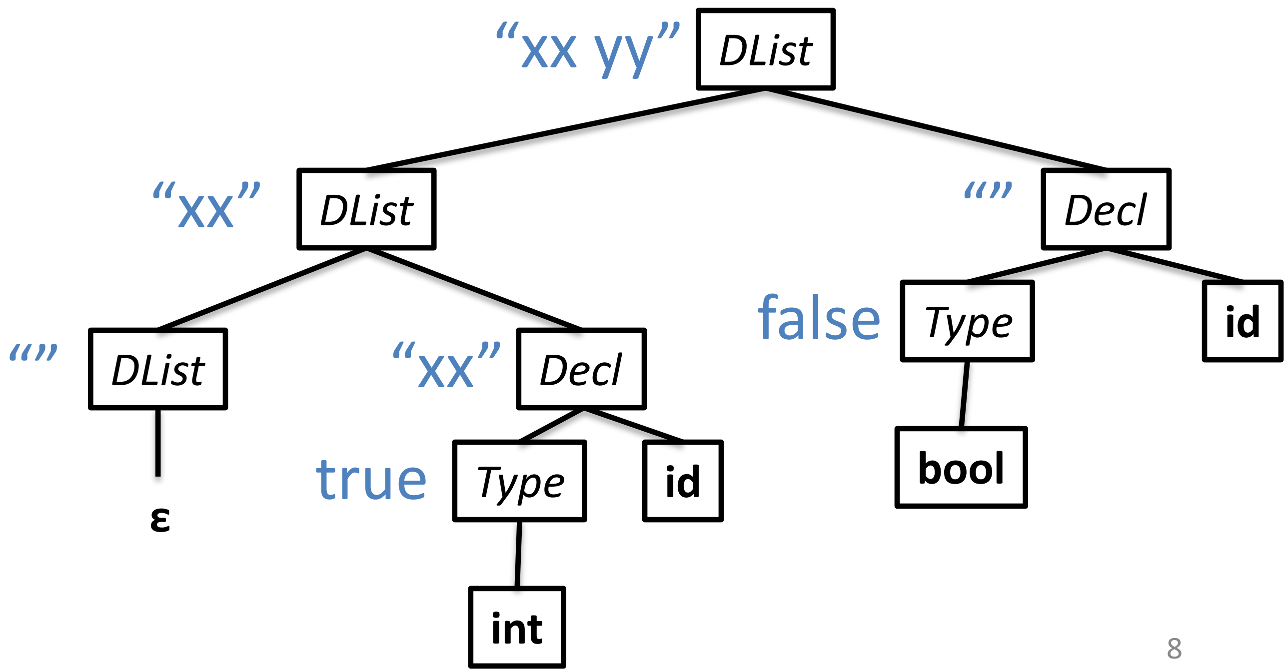
CFG			Rules
DList	→	**ε**	*DList*.trans = “”
	|	*Decl DList*	*DList.trans* = *Decl.trans* + “ “ + $DList_2$.trans
Decl	→	*Type* **id ;**	*Decl*.trans = **id**.value
Type	→	**int**	
	|	**bool**	

SDT Example 2b: ints only

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

CFG			Rules
DList	→	**ε**	*DList*.trans = “”
	|	*Decl DList*	*DList.trans* = *Decl.trans* + “ “ + DList_2.trans
Decl	→	*Type* **id ;**	if (type.trans) {*Decl*.trans = **id**.value} else {*Decl*.trans = “”}
Type	→	**int**	*Type*.trans = true
	|	**bool**	*Type*.trans = false

Input string
int xx;
bool yy;



Different nonterms can have different types

Rules can have conditionals

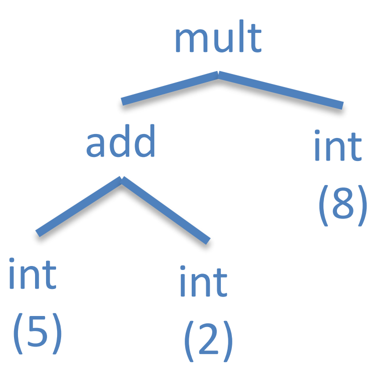
SDT for Parsing

- 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)

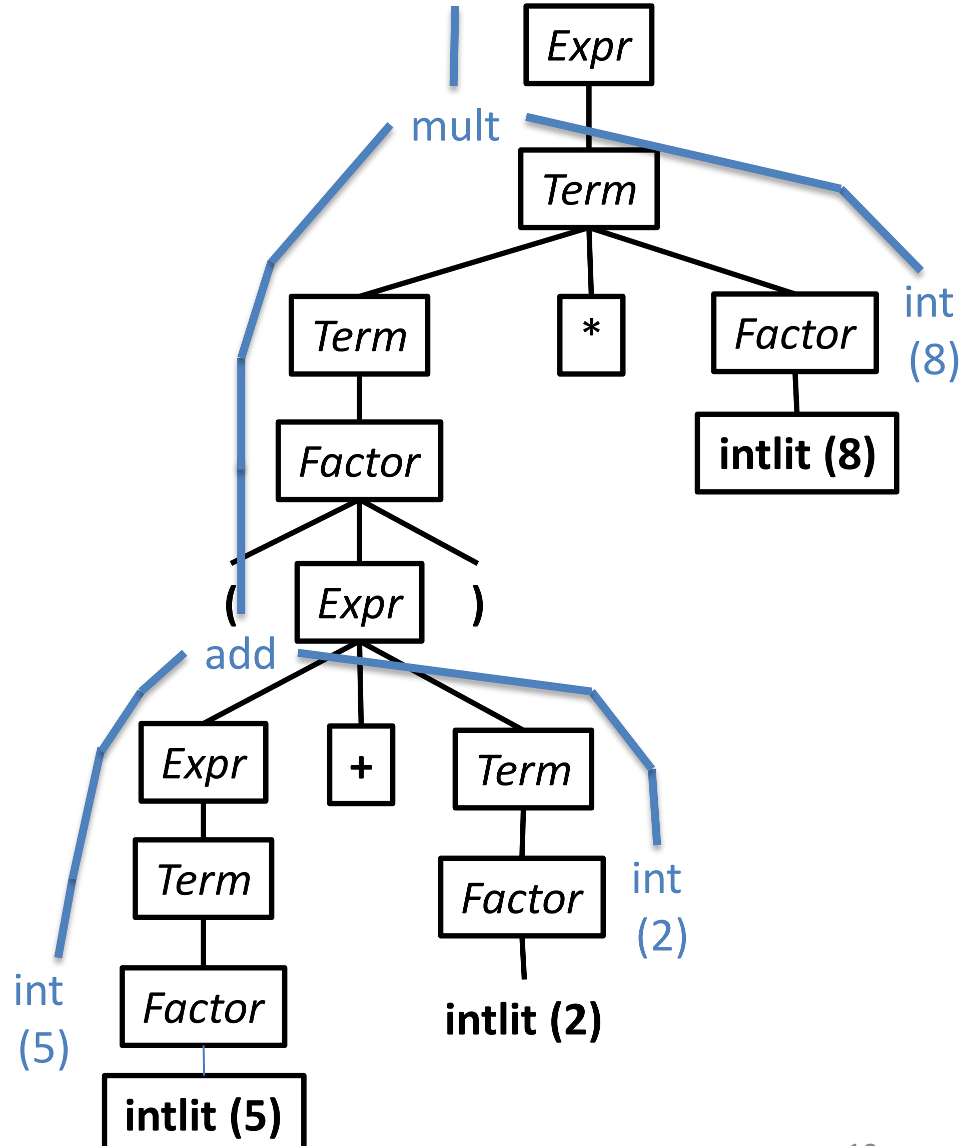
Abstract Syntax Trees

- 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 )
```

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

AST Implementation

- How do we actually represent an AST in code?
- We'll take inspiration from how we represented tokens in JLex

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 define DList to be represented as

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

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

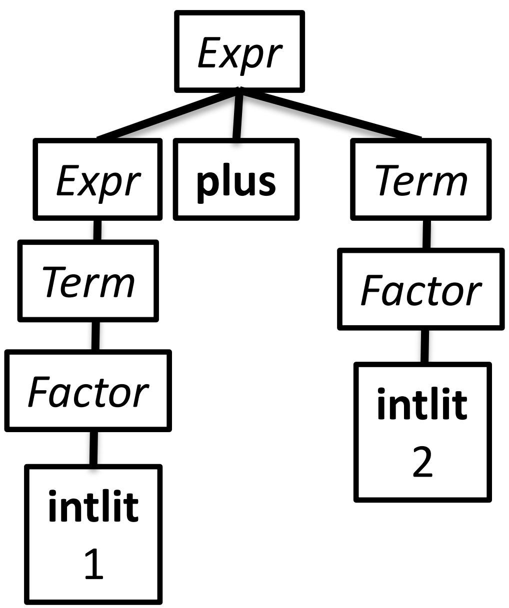
Thinking about implementing ASTs

- Consider the AST for a simple language of Expressions

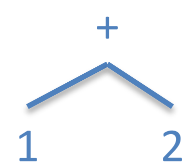
<u>Input</u>
1 + 2

<u>Tokenization</u>
intlit plus intlit

<u>Parse Tree</u>



<u>AST</u>



<u>Naïve AST Implementation</u>

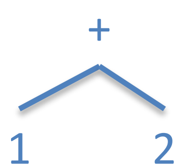
```
class PlusNode
        IntNode left;
        IntNode right;
}

class IntNode{
        int value;
}
```

Thinking about implementing ASTs

- Consider AST node classes
  – We'd like the classes to have a common inheritance tree

AST

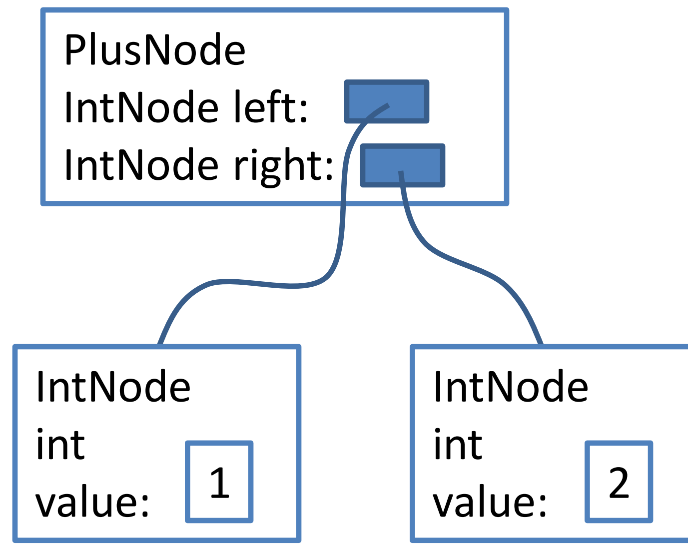


Naïve AST Implementation

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

class IntNode
{       int value;
}
```

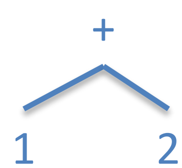
Naïve java AST



Thinking about implementing ASTs

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

AST



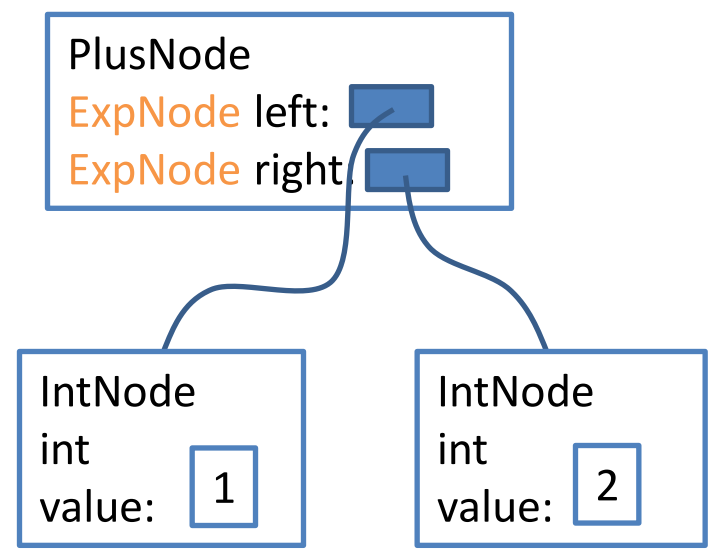
Naïve AST Implementation

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

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

Make these extend ExpNode

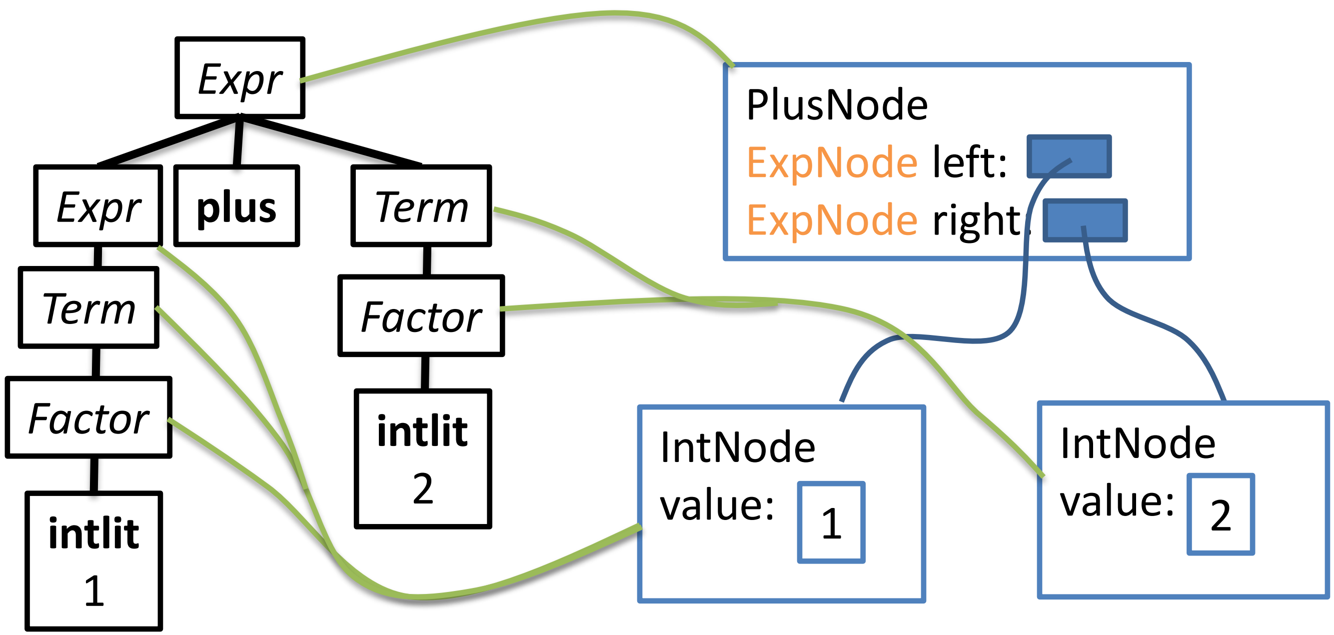
Better java AST



Implementing ASTs for Expressions

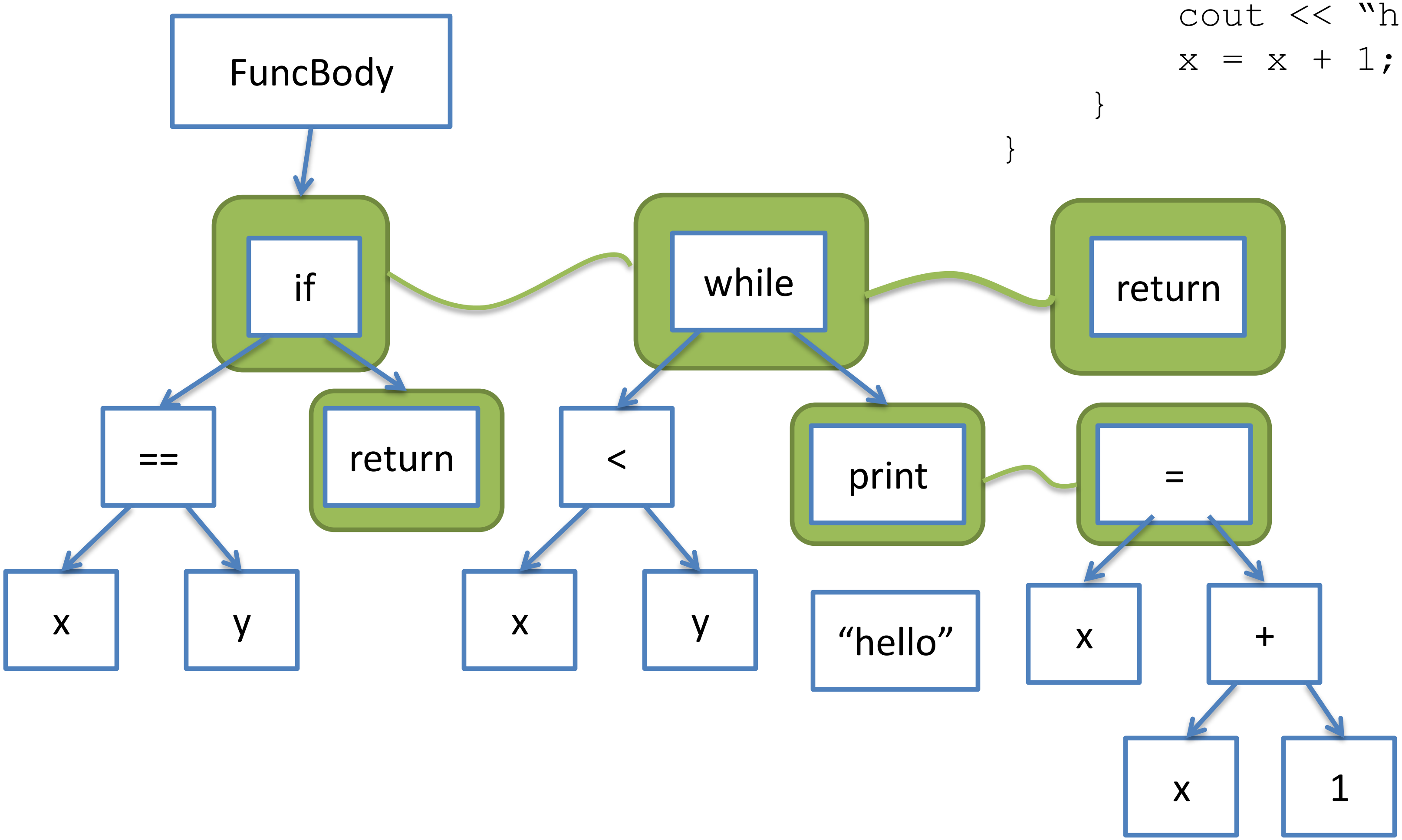
CFG			Translation Rules
Expr	->	Expr + Term	*Expr1*.trans = new PlusNode(*Expr2*.trans, *Term*.trans)
	|	Term	*Expr*.terms = Term.trans
Term	->	Term * Factor	*Term1*.trans = new TimesNode(Term2.trans, *Factor*.trans)
	|	Factor	*Term*.trans = *Factor*.trans
Factor	->	intlit	*Factor*.trans = new IntNode(**intlit**.value)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

Example: 1 + 2



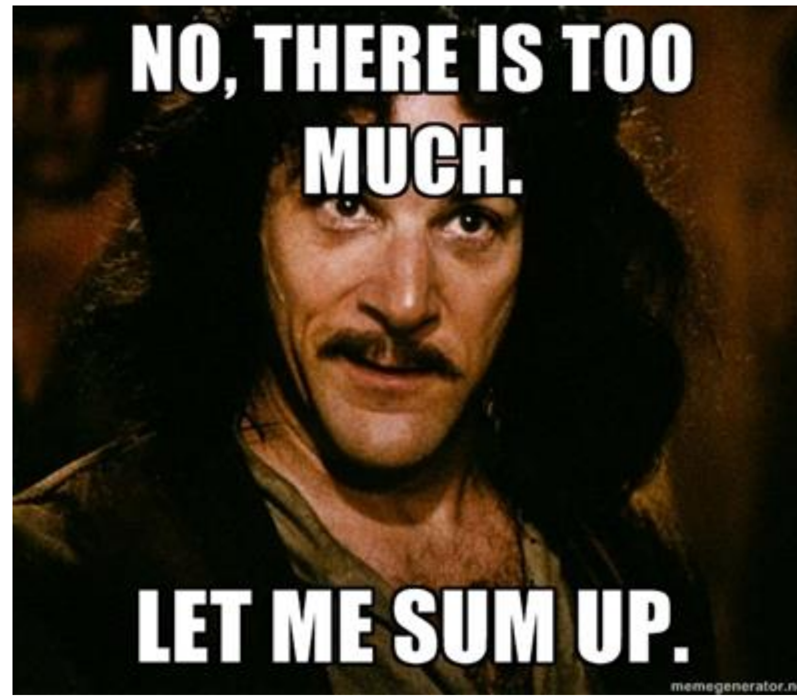
An AST for a C Flat Snippet

```
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 ← Next time
Implementation: ??? ← Next week