Checkin 6

Assign SDD rules to each of the following rules according to the following goal:

Goal: Each B translates to whether the subtree has an odd number of “1”s

```
B ::= B 0
    | B 1
    | ε
```

Announcements

Administrivia

Quiz 1 on Friday next week

Covers material up to and including today's lecture

Review Session TBA

No labs next week

Labor day

University of Kansas | Drew Davidson

EECS 665

COMPILER CONSTRUCTION

Abstract Syntax Trees

Last Time

Review: Syntax-Directed Translation

Working with Parse Trees

- Perspectives on trees
  - Data structure / recursive definition
- Syntax-Directed Definitions
  - Specifies *meaning* of parse trees
    - Digit sequence as integer
    - Expression as its evaluation
    - Expression as its operator count

$\$\$$

$\$1$ $\$2$ $\$3$

$E ::= E \text{ plus } T$

$RHS.trans = LHS_1.trans + LHS_3.trans$

Syntax-Directed Definition

Last Time

Review: Syntax-Directed Translation

Enrich a Context-Free Grammar with a set of translation rules

- Rules are the Syntax-Directed Definition
- Applying those rules creates a Syntax-Directed Translation

Today's Outline

Abstract Syntax Trees

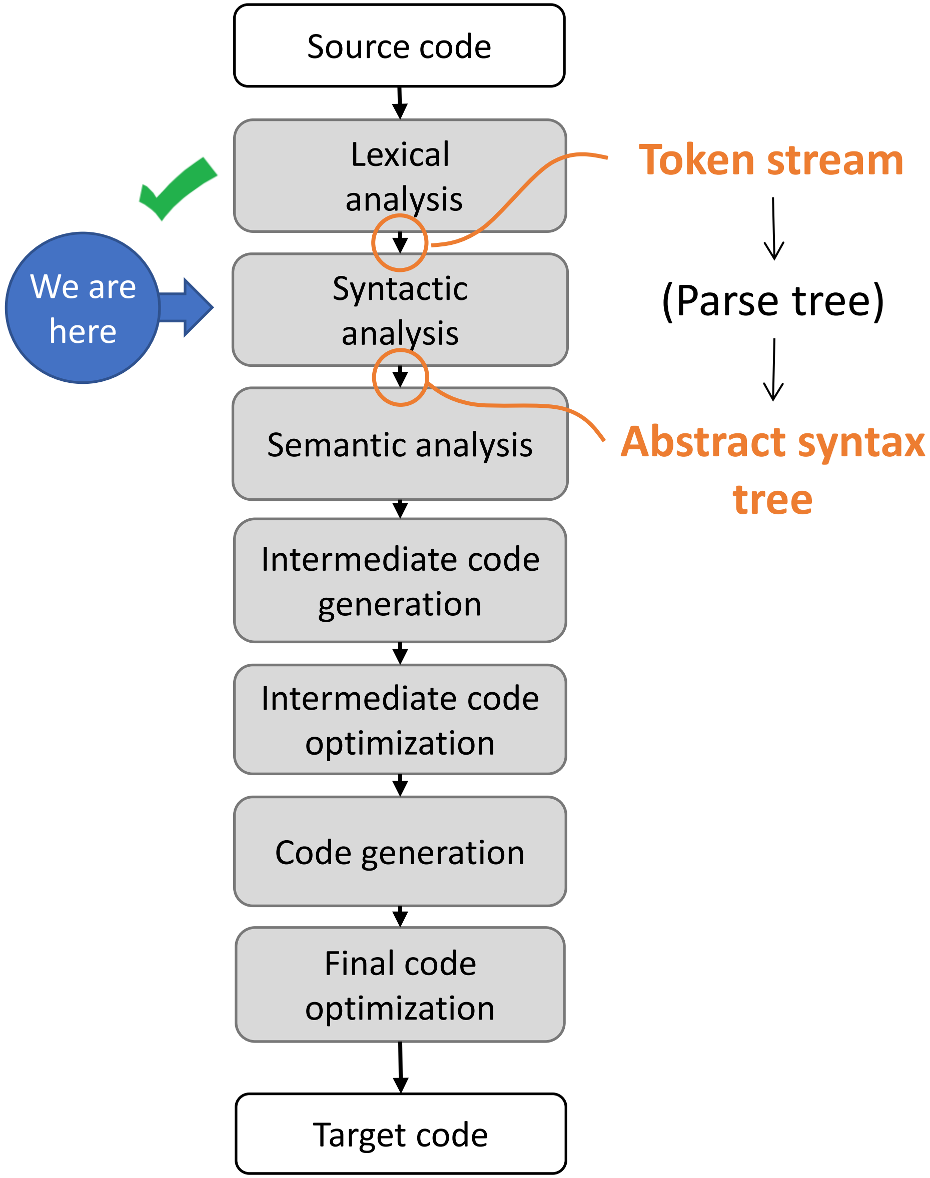
Abstract Syntax Trees

- Context for ASTs
- ASTs for Programs
- ASTs in Code

Syntax-Directed Definition

Compiler Construction

Progress Pics



We have most of the concepts of syntactic analysis

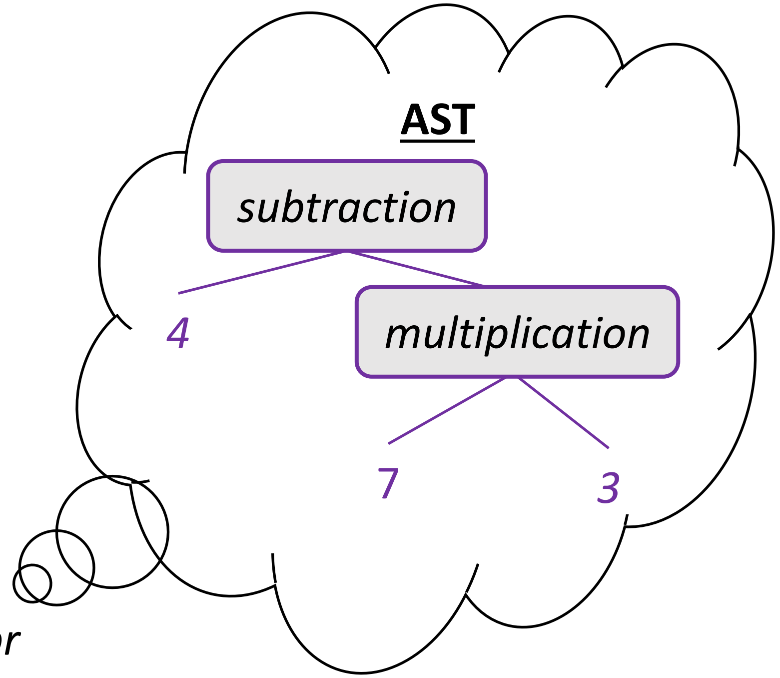
Syntactic Analysis

- Output: AST
- Specification:
  - Syntax: CFG
  - Semantics: SDD

just need to apply the concepts to a programming language

The Parse Tree vs AST

Abstract Syntax Trees



Parse tree

- Recognizes valid syntax
- Nodes are rules that constrain structure

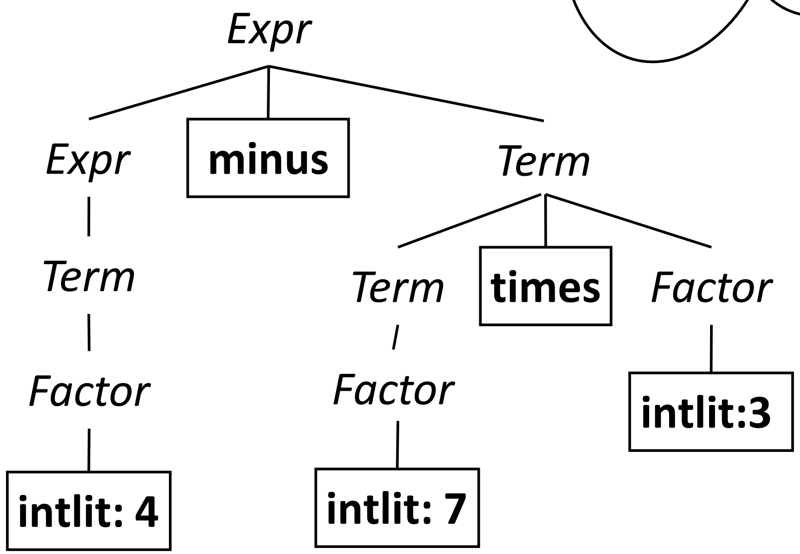
AST

- Transcends concrete syntax
- Represents program semantics
- Nodes are language concepts

Flat expression

`4 – 7 * 3`

Parse tree



AST: Implicit in the Parse Tree

Abstract Syntax Trees

Parse tree implies an AST

- Syntactic analysis “extracts” the AST from the parse tree
- Conceptually, a post-processing pass over the parse tree

A tree within a tree

AST: Implicit in the Parse Tree

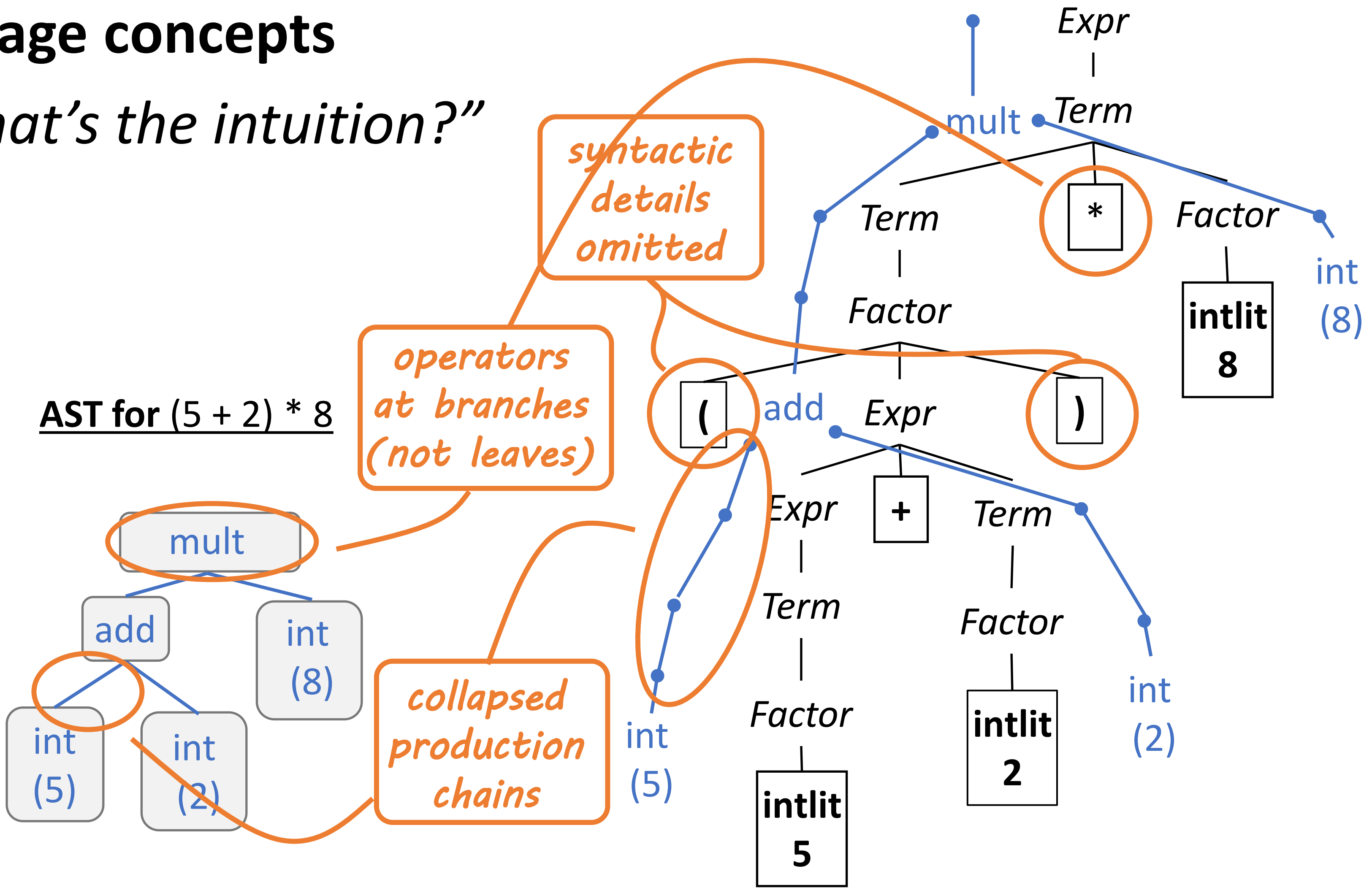
Abstract Syntax Trees

Nonterminals map to language concepts

- *“What’s the intuition?”*

Parse Tree for (5+2)*8

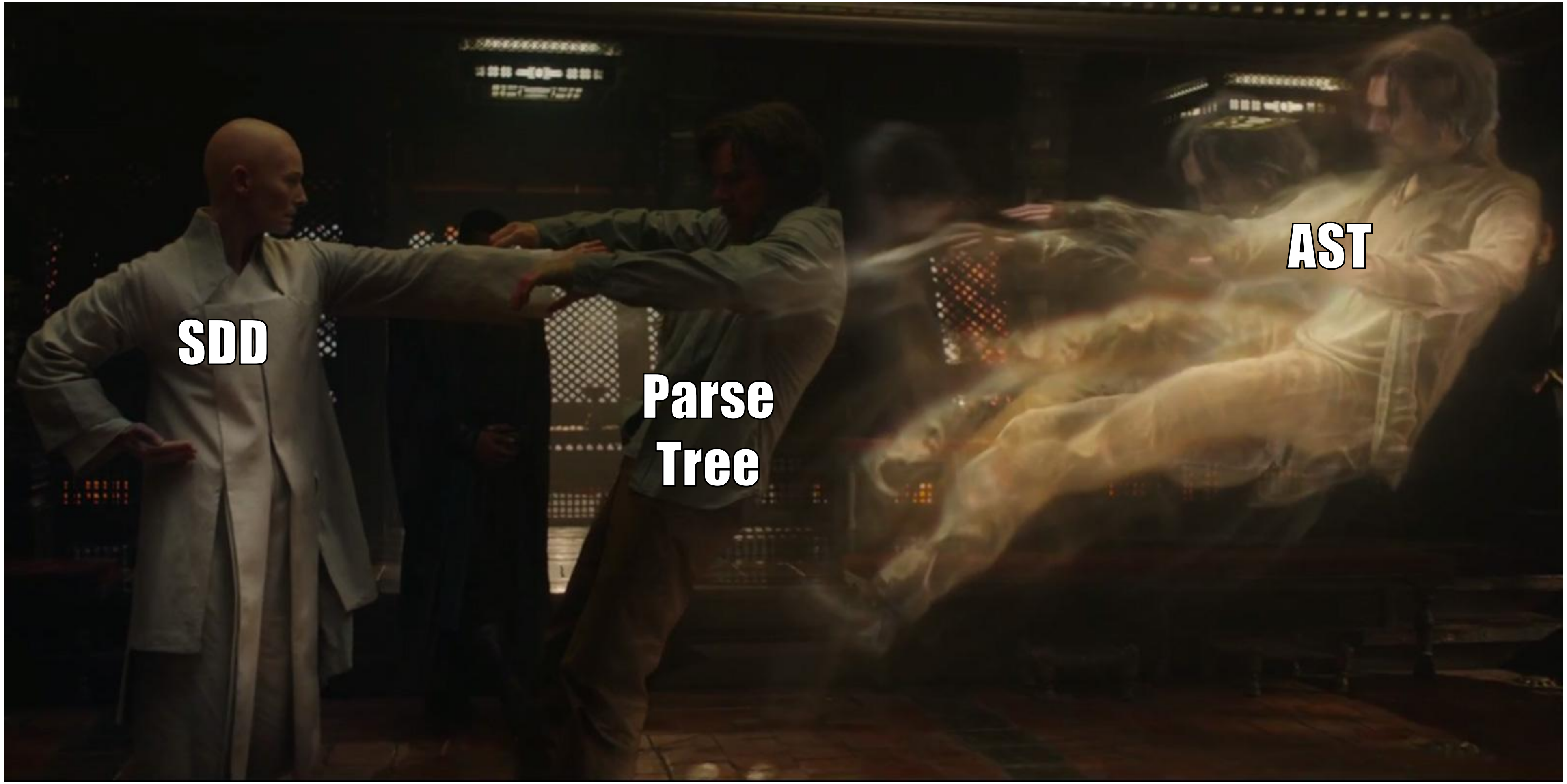
AST for (5 + 2) * 8



From Parse Tree to AST

Abstract Syntax Trees

SDD knocks the AST out of the parse tree

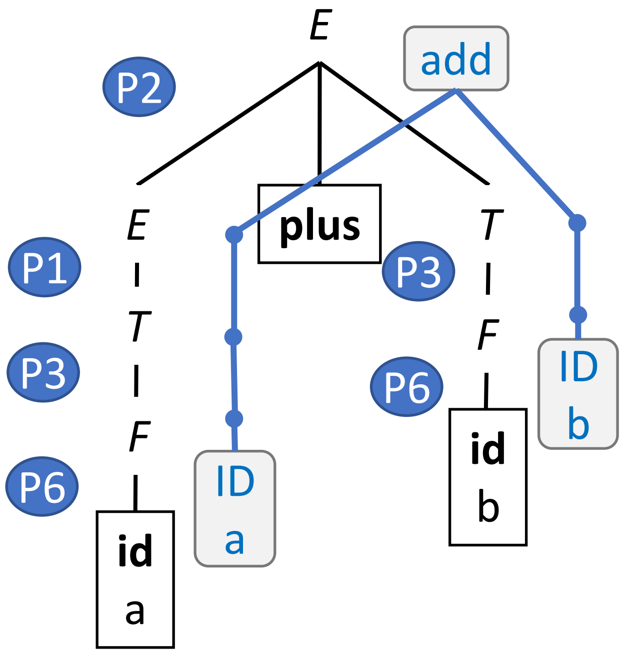


Defining AST Structure

Abstract Syntax Trees

	CFG Production	AST Building Rule (Pseudocode)
P1	*E* ::= *T*	*LHS*.trans = *T*.trans
P2	*E* ::= *E* **plus** *T*	*LHS*.trans = AddNode where AddNode.arm1 = *E*.trans AddNode.arm2 = *T*.trans
P3	*T* ::= *F*	*LHS*.trans = *F*.trans
P4	*T* ::= *T* **times** *F*	*LHS*.trans = MultNode where MultNode.arm1 = *T*.trans MultNode.arm2 = *F*.trans
P5	*F* ::= **lpar** *E* **rpar**	*LHS*.trans = *E*.trans
P6	*F* ::= **id**	*LHS*.trans = IDNode where IDNode.content = **id**.value

Input String: `a + b`

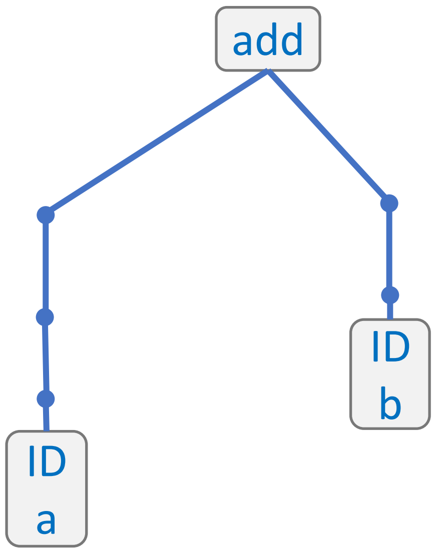


Defining AST Structure

Abstract Syntax Trees

	CFG Production	AST Building Rule (Pseudocode)
P1	*E* ::= *T*	*LHS*.trans = *T*.trans
P2	*E* ::= *E* **plus** *T*	*LHS*.trans = AddNode where AddNode.arm1 = *E*.trans AddNode.arm2 = *T*.trans
P3	*T* ::= *F*	*LHS*.trans = *F*.trans
P4	*T* ::= *T* **times** *F*	*LHS*.trans = MultNode where MultNode.arm1 = *T*.trans MultNode.arm2 = *F*.trans
P5	*F* ::= **lpar** *E* **rpar**	*LHS*.trans = *E*.trans
P6	*F* ::= **id**	*LHS*.trans = IDNode where IDNode.content = **id**.value

Input String: `a + b`



Today's Outline
Abstract Syntax Trees

Abstract Syntax Trees

- Context for ASTs
- ASTs for Programs
- ASTs in Code

Syntax-Directed Definition

Capturing Program Constructs

Abstract Syntax Trees – ASTs for Programs

ASTs can (and must) capture all program constructs

- Simply need to attach the right SDD rule to each production
- May need to tweak the grammar a bit
  - (we already saw this with syntactic ambiguity)

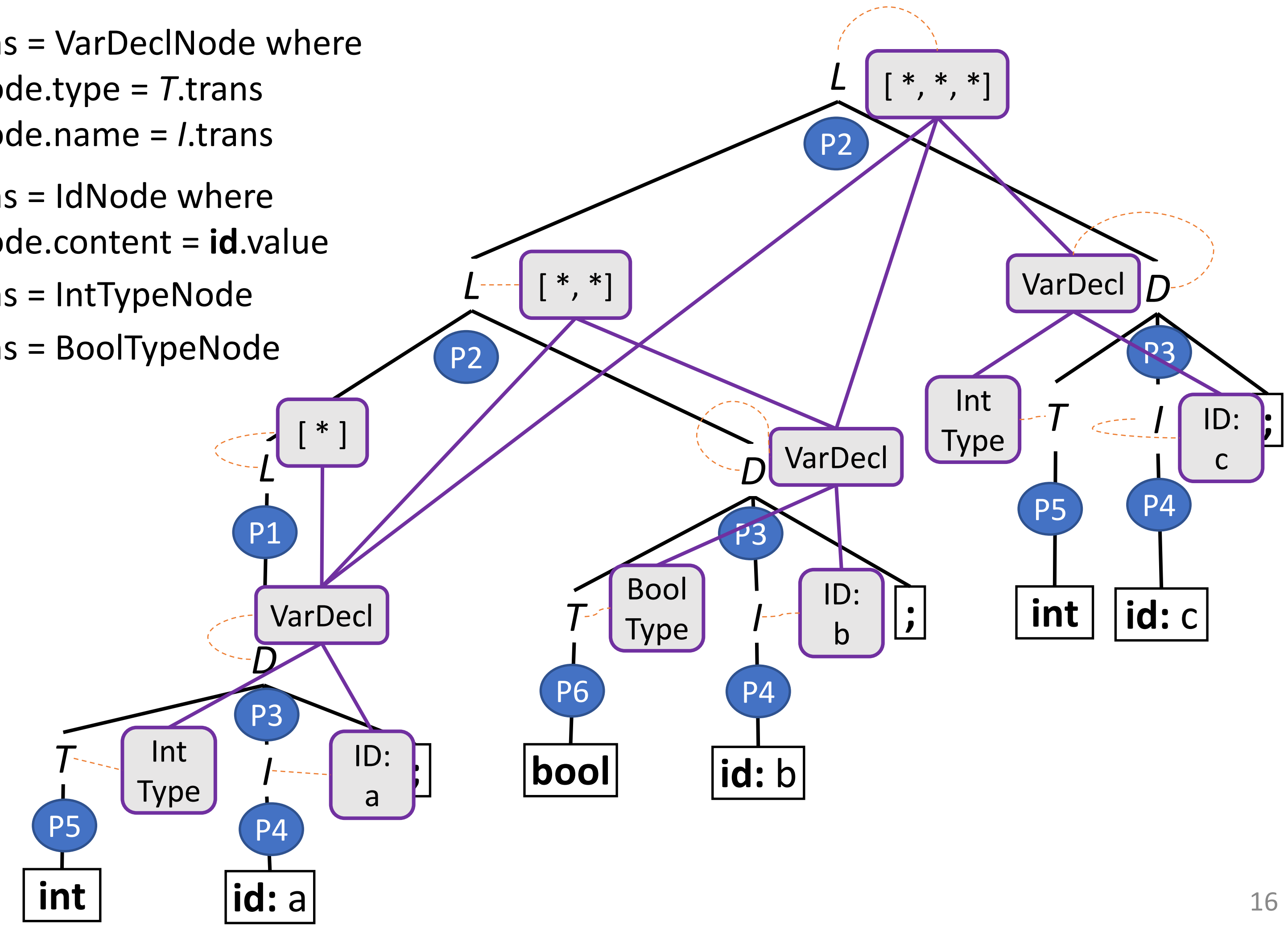
Capturing Program Constructs

Abstract Syntax Trees – ASTs for Programs

P1	*L* ::= *D*	*LHS*.trans = [*D*.trans]
P2	| *L D*	*LHS*.trans = [].addAll(*L*.trans).add(*D*.trans)
P3	*D* ::= *T I* **;**	*LHS*.trans = VarDeclNode where Node.type = *T*.trans Node.name = *I*.trans
P4	*I* ::= **id**	*LHS*.trans = IdNode where Node.content = **id**.value
P5	*T* ::= **int**	*LHS*.trans = IntTypeNode
P6	| **bool**	*LHS*.trans = BoolTypeNode

Input String:

```
int a; bool b; int c;
```



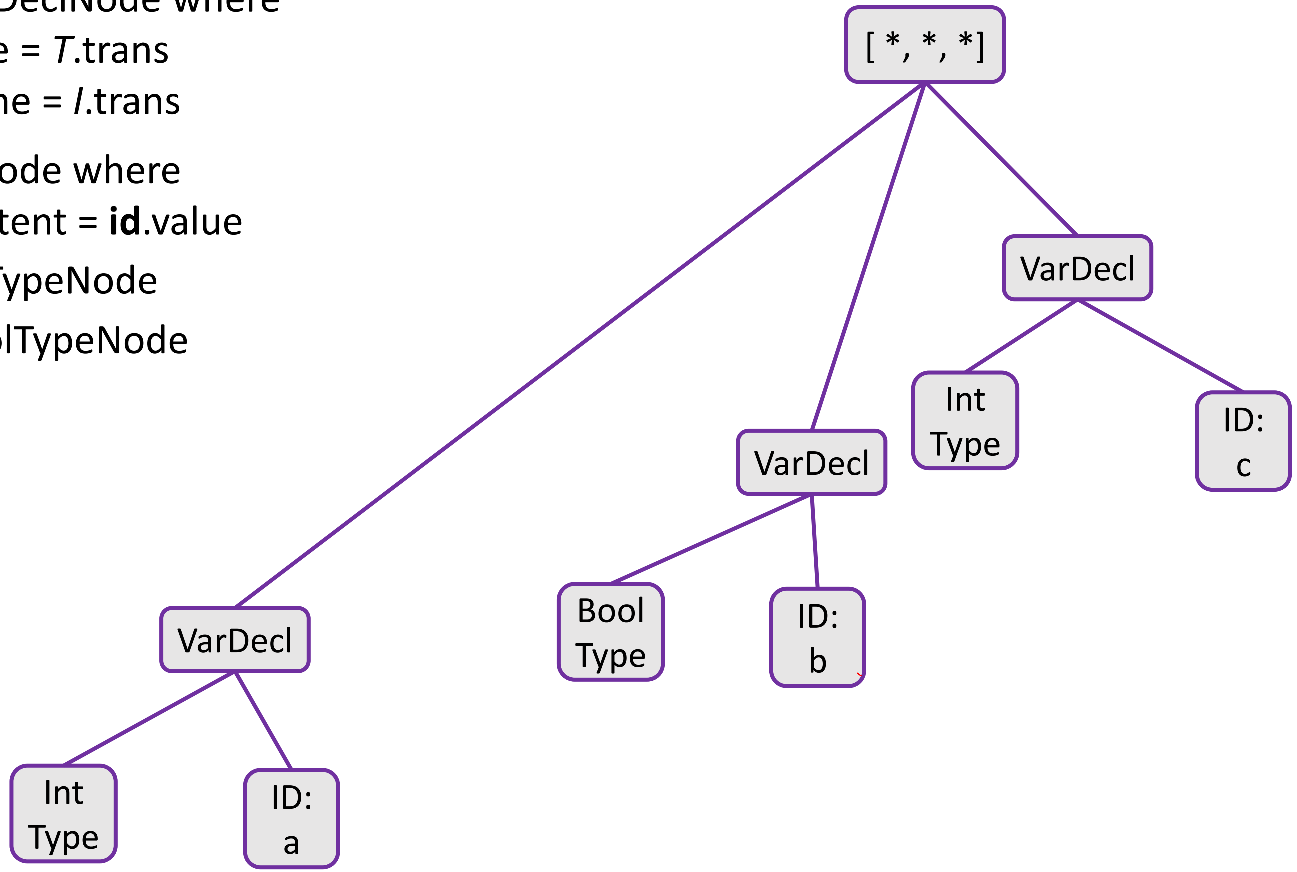
Capturing Program Constructs

Abstract Syntax Trees – ASTs for Programs

P1	*L* ::= *D*	*LHS*.trans = [*D*.trans]
P2	| *L D*	*LHS*.trans = [].addAll(*L*.trans).add(*D*.trans)
P3	*D* ::= *T I* **;**	*LHS*.trans = VarDeclNode where Node.type = *T*.trans Node.name = *I*.trans
P4	*I* ::= **id**	*LHS*.trans = IdNode where Node.content = **id**.value
P5	*T* ::= **int**	*LHS*.trans = IntTypeNode
P6	| **bool**	*LHS*.trans = BoolTypeNode

Input String:

```
int a; bool b; int c;
```



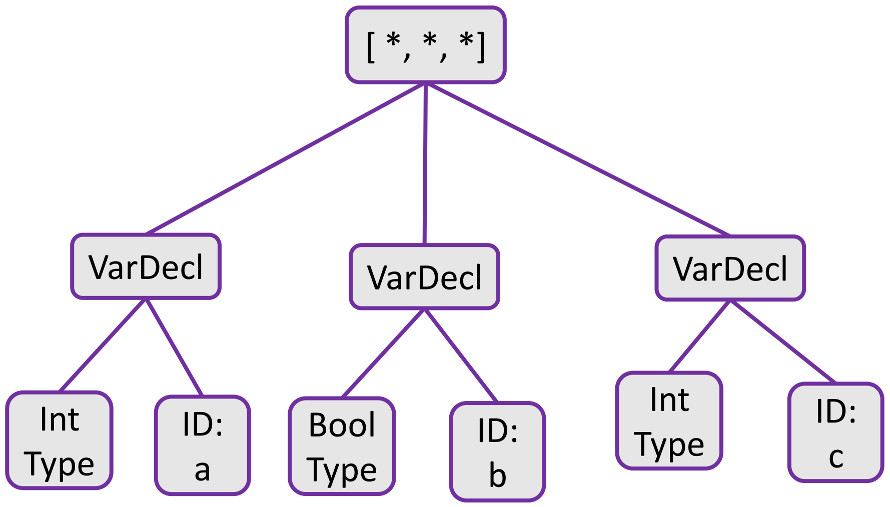
Capturing Program Constructs

Abstract Syntax Trees – ASTs for Programs

P1	*L* ::= *D*	*LHS*.trans = [*D*.trans]
P2	| *L D*	*LHS*.trans = [].addAll(*L*.trans).add(*D*.trans)
P3	*D* ::= *T I* **;**	*LHS*.trans = VarDeclNode where Node.type = *T*.trans Node.name = *I*.trans
P4	*I* ::= **id**	*LHS*.trans = IdNode where Node.content = **id**.value
P5	*T* ::= **int**	*LHS*.trans = IntTypeNode
P6	| **bool**	*LHS*.trans = BoolTypeNode

Input String:

`int a; bool b; int c;`



AST: much more intuitive program representation

Today's Outline

Abstract Syntax Trees

Abstract Syntax Trees

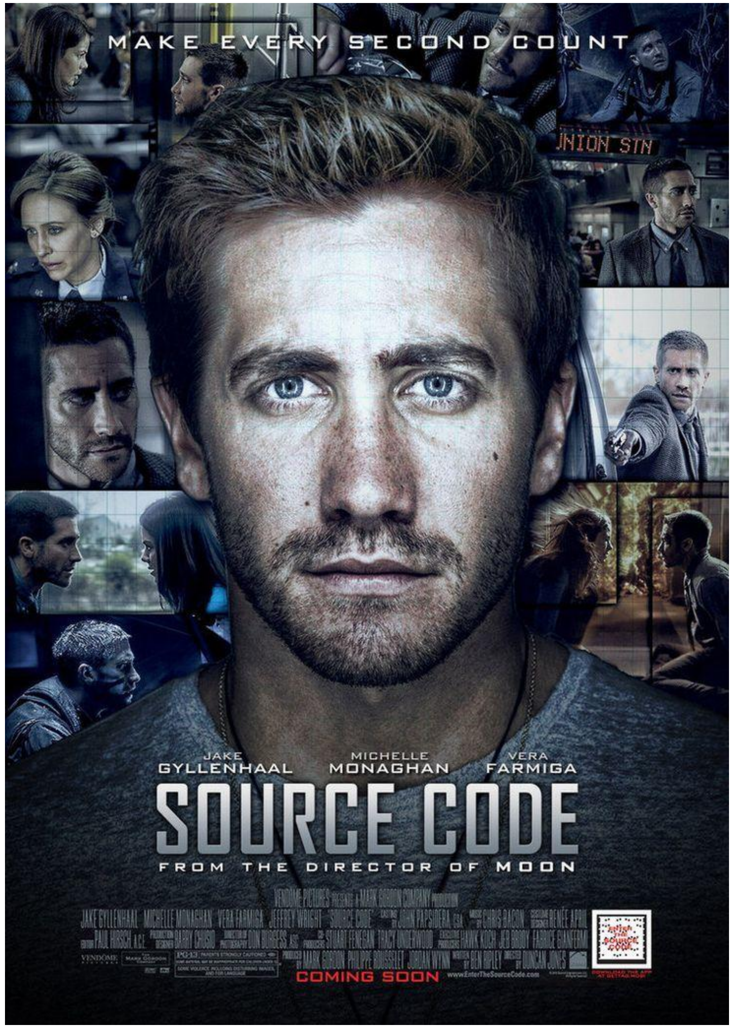
- Context for ASTs
- ASTs for Programs
- ASTs in Code

Syntax-Directed Definition

What do ASTs Look like *in Code*?

Abstract Syntax Trees – ASTs in Code

- Let’s contemplate source code to implement ASTs



Implementing AST Nodes

Abstract Syntax Trees – ASTs in Code

We've given AST nodes pseudocode properties

- We'll lean into that with C++ classes and members

```
class DeclNode{
    TypeNode& type;
    IDNode& name;
};
```

<u>CFG Production</u> **<u>Rule</u>**

Decl ::= *Type ID* **;** *LHS*.trans = DeclNode where

.type = *Type*.trans

.name = *ID*.trans

```
class IDNode{
    char * content;
};
```

ID ::= **id** *LHS*.trans = IDNode where

.content = **id**.value

Recall OOP: is-a vs has-a

Abstract Syntax Trees – ASTs in Code

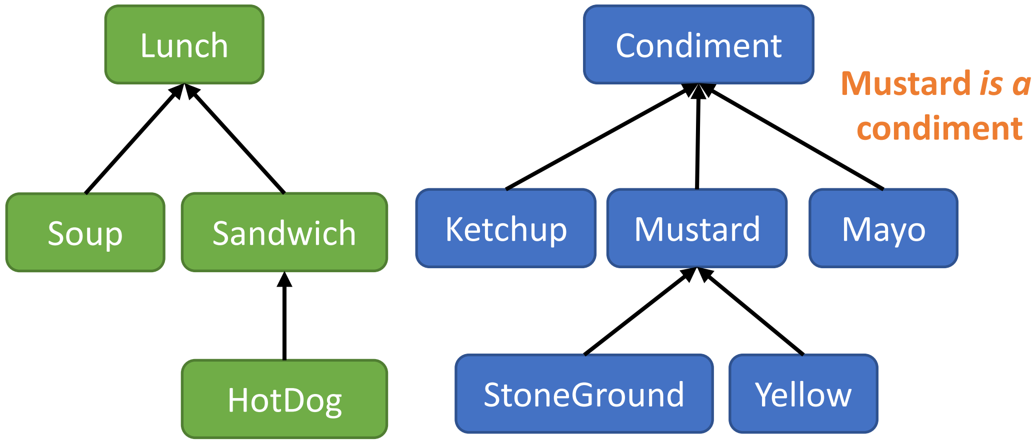
```
class Lunch  {
public:
  void eat(){ … }
};

class HotDog : public Sandwich{
public:
  Ketchup * myK;
  Mustard * myM;
};
```

```
int main(){
  HotDog * h = new HotDog();
  Mustard * m = new Yellow();
  h->myM = m;
  h->eat();
  h->myM = k;  (error)
}
```

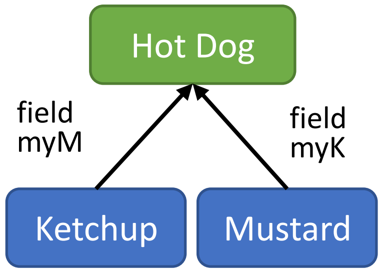
A hot dog *is a* sandwich

A hot dog *is a* lunch

Inheritance (*is-a* relation)



Components (*has-a* relation)



A hot dog *has a* mustard topping

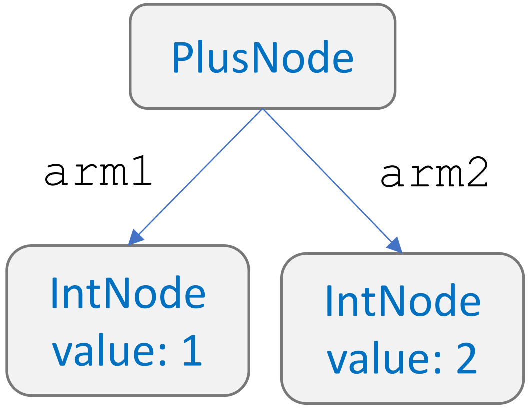
ASTs In Code: Connecting Subtrees

Naïve
AST Implementation

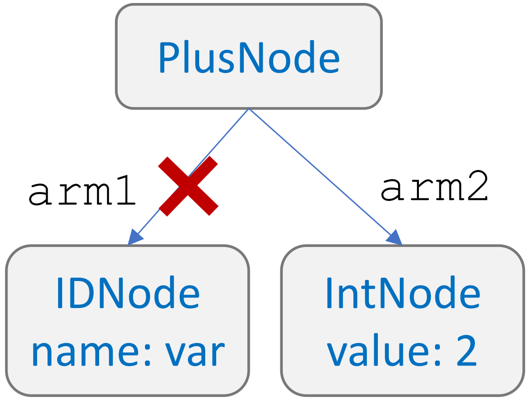
Oops! Disallows Variables in expressions!

```
class PlusNode
{   IntNode& arm1;
    IntNode& arm2;
};

class IntNode
{   int value; };

class IDNode
{   std::string content; };
```

AST for `1+2`



AST for `var+2`



ASTs: Connecting Subtrees

Abstract Syntax Trees – ASTs in Code

Naïve

<u>AST Implementation</u>

```
class ASTNode{ … };

class PlusNode : public ASTNode
{   IntNode& left;
    IntNode& right;      ASTNode
};

class IntNode : public ASTNode
{   int value; };

class IDNode : public ASTNode
{   std::string content; };


class DeclNode : public ASTNode
{ … }
```

<u>Class hierarchy for AST Nodes</u>

ASTNode
- IntNode
- IDNode
- DeclNode
- ⋮

ASTs: Connecting Subtrees

Abstract Syntax Trees – ASTs in Code

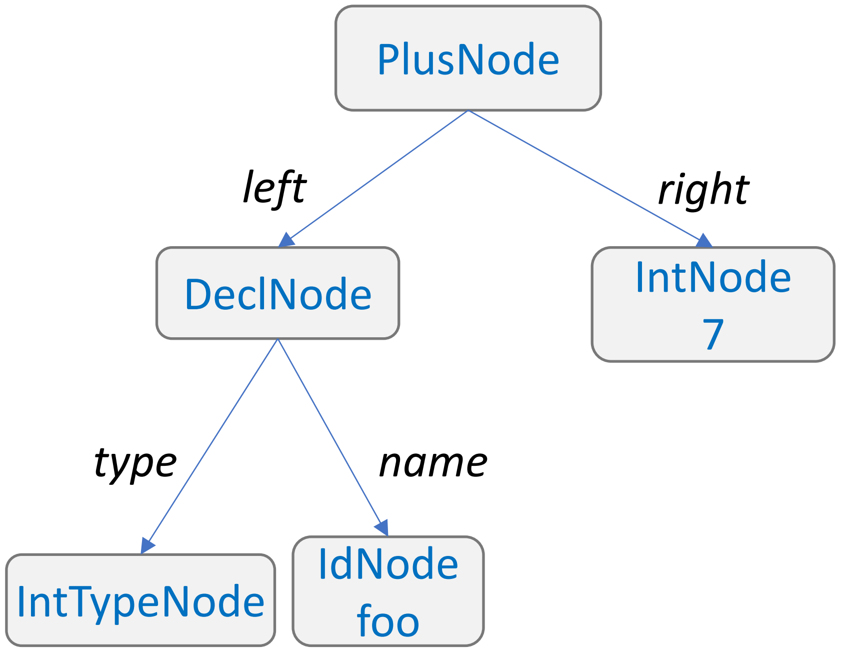
Less Naïve ~~Naïve~~ AST Implementation

```
class ASTNode{ };
```

Wrongly allows all node types! Too generic!

```
class PlusNode : public ASTNode
{   ASTNode& left;
    ASTNode& right;
};

class IntNode : public ASTNode
{   int value; };

class IDNode : public ASTNode
{   std::string content; };


class DeclNode : public ASTNode
{  … }
```

`int foo; + 7`



ASTs: Connecting Subtrees

Abstract Syntax Trees – ASTs in Code

Ok

~~Less Naïve~~

~~Naïve~~

<u>AST Implementation</u>

`class ASTNode{ };`

`class ExprNode : public ASTNode {};`

`class PlusNode : public` ~~`ASTNode`~~
`{` ~~`ASTNode`~~`& left;`
~~`ASTNode`~~`& right;`
`};`

ExprNode

`class IntNode : public` ~~`ASTNode`~~
`{ int value; };`

ExprNode

`class IDNode : public` ~~`ASTNode`~~
`{ std::string content; };`

ExprNode

`class DeclNode : public ASTNode`
`{ … }`

Solution:
Proper line of succession

<u>Class hierarchy for AST Nodes</u>

- ASTNode
  - ExprNode
    - IntNode
    - IDNode
    - ⋮
  - DeclNode
  - ⋮

Crafting the AST Hierarchy

Abstract Syntax Trees – ASTs in Code

Which classes should inherit which other classes?

- No one “right way” to do it

Summary
Abstract Syntax Trees

The Abstract Syntax Tree

- A lightweight abstraction of the program syntax
- A data structure in the compiler's host language
  - Lexical Tokens: Flex Regular Expression Rules
  - Parse Tree: Bison Context-Free Grammar Rules
  - AST: Back to C++ objects and classes