

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

$$\begin{array}{lcl} B & ::= & B \ 0 \\ & | & B \ 1 \\ & | & \epsilon \end{array}$$

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

$$\begin{array}{lcl} B & ::= & B \ 0 \\ & | & B \ 1 \\ & | & \epsilon \end{array}$$

Announcements

Administrivia

Quiz 1 on Friday

Covers material up to and including
today's lecture

ECCS 665
COMPILER
CONSTRUCTION

The title is presented in a highly stylized, futuristic font. The words 'COMPILER' and 'CONSTRUCTION' are in large, bold, metallic letters with a glowing blue and purple outline. Above them, 'ECCS 665' is written in a smaller, pink, cursive-like font. The background is a dark blue space with stars and a glowing purple grid pattern.

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



**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

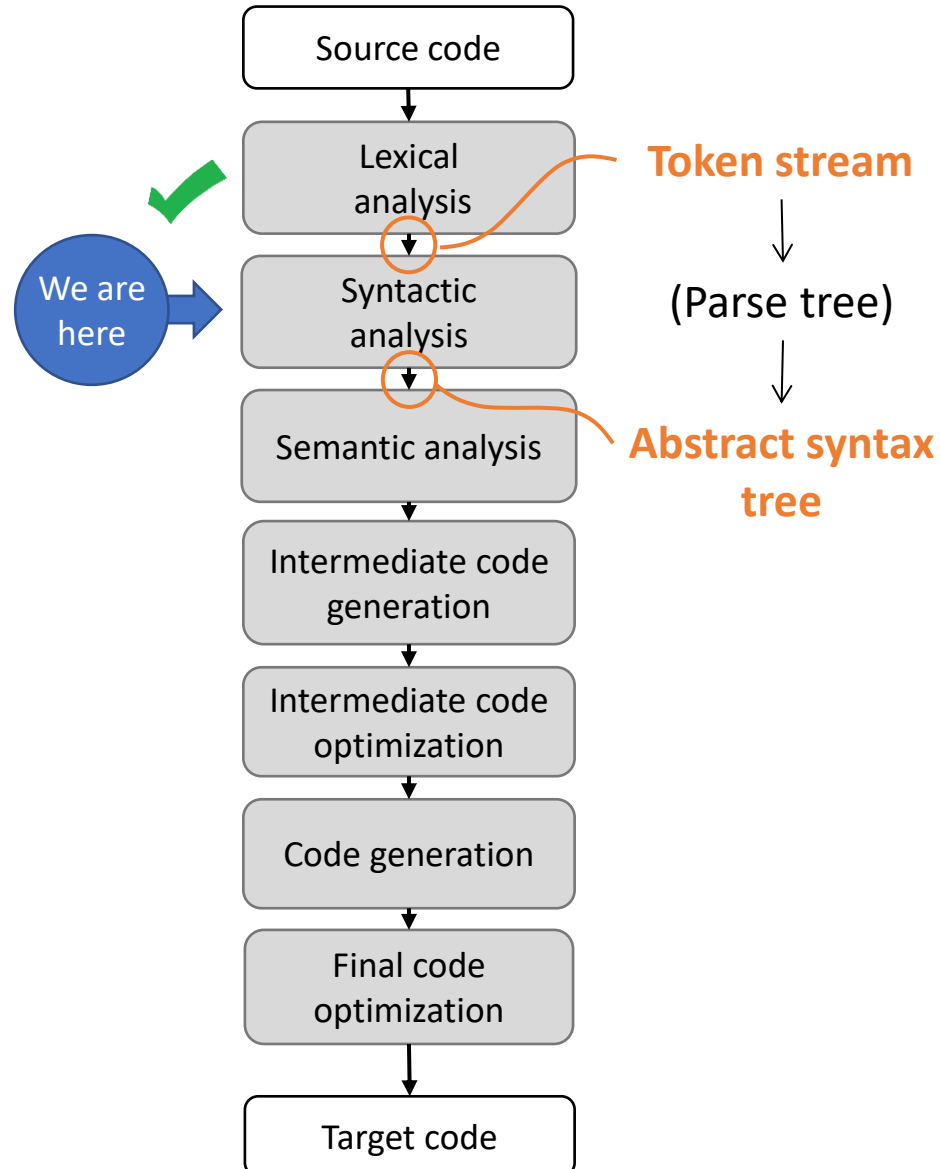
- Context for ASTs
- ASTs for Programs
- ASTs in Code



**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

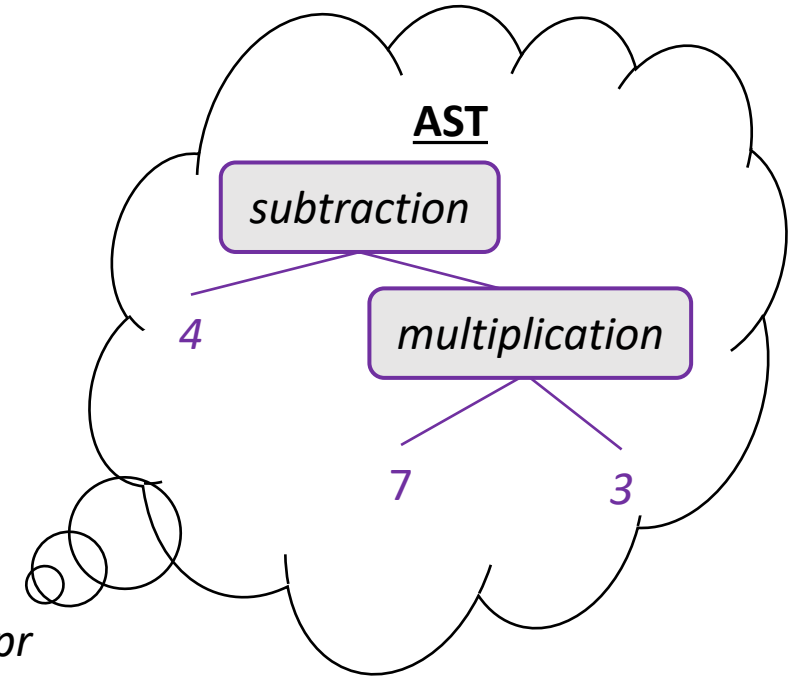
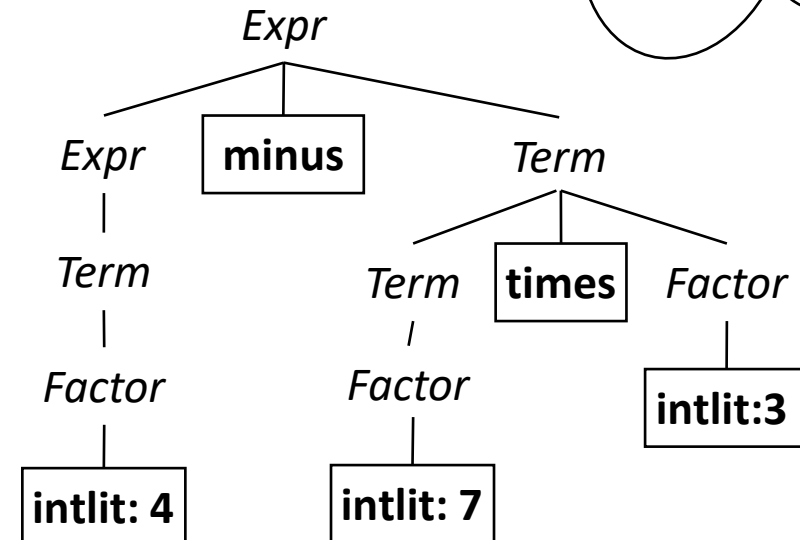
AST

- 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

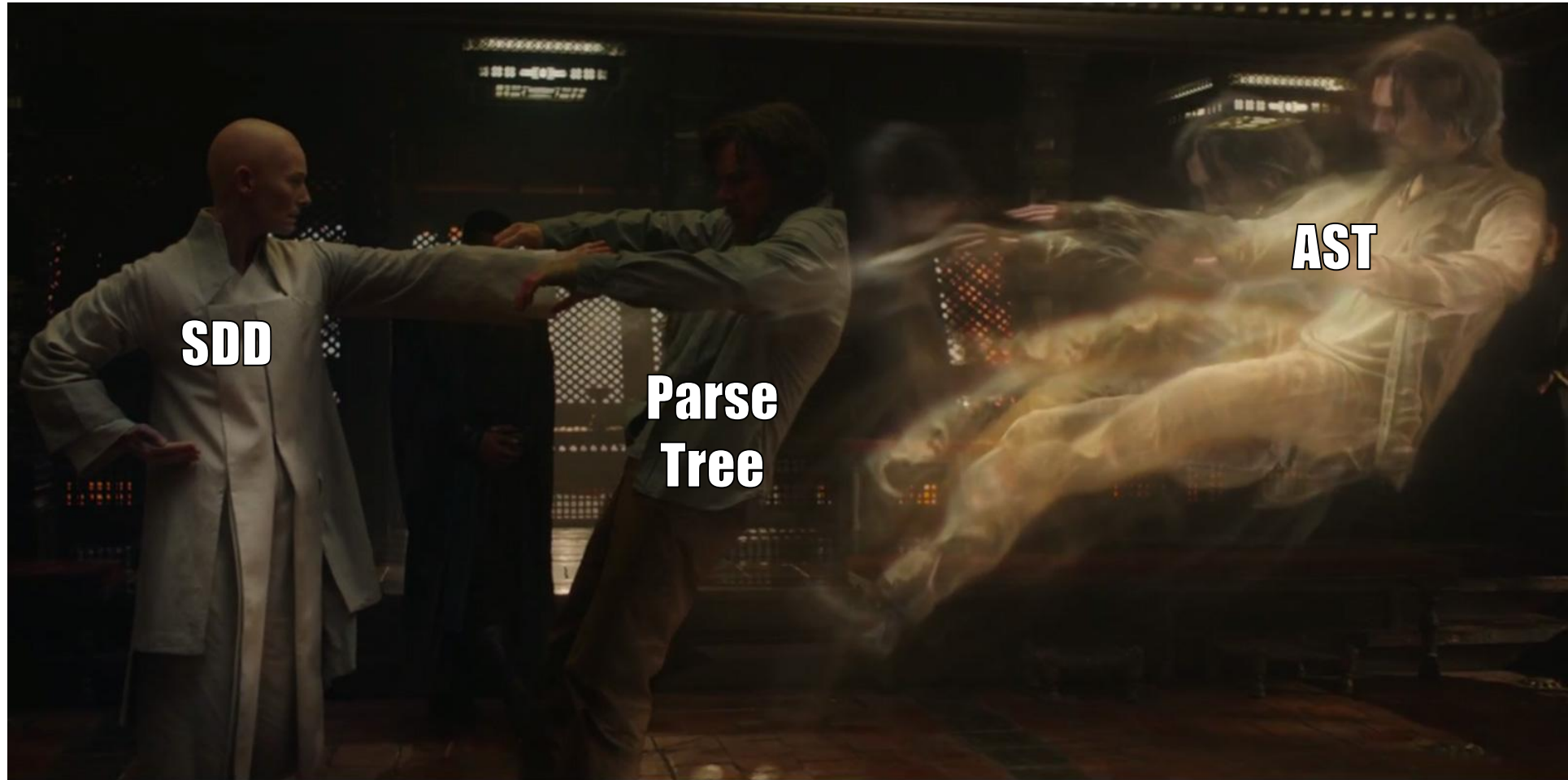


A tree within a tree

From Parse Tree to AST

Abstract Syntax Trees

SDD knocks the AST out of the parse tree

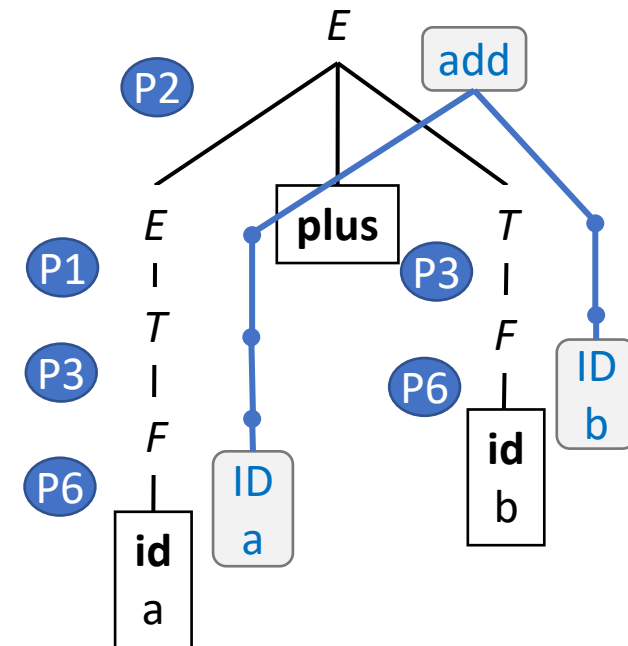


Defining AST Structure

Abstract Syntax Trees

<u>CFG Production</u>	<u>AST Building Rule (Pseudocode)</u>
P1 $E ::= T$	$LHS.trans = T.trans$
P2 $E ::= E \text{ plus } T$	$LHS.trans = \text{AddNode}$ where $\text{AddNode.arm1} = E.trans$ $\text{AddNode.arm2} = T.trans$
P3 $T ::= F$	$LHS.trans = F.trans$
P4 $T ::= T \text{ times } F$	$LHS.trans = \text{MultNode}$ where $\text{MultNode.arm1} = T.trans$ $\text{MultNode.arm2} = F.trans$
P5 $F ::= \text{lpar } E \text{ rpar}$	$LHS.trans = E.trans$
P6 $F ::= \text{id}$	$LHS.trans = \text{IDNode}$ where $\text{IDNode.content} = \text{id.value}$

Input String: a + b

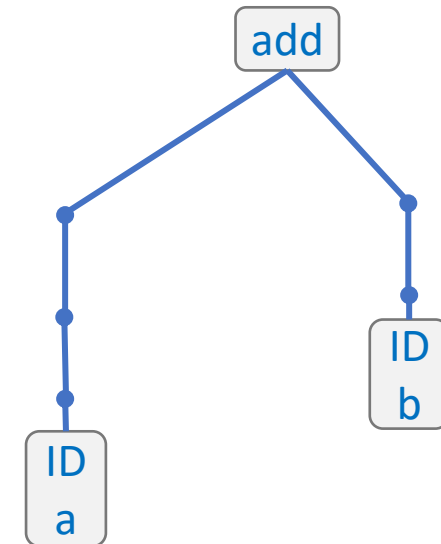


Defining AST Structure

Abstract Syntax Trees

<u>CFG Production</u>	<u>AST Building Rule (Pseudocode)</u>
P1 $E ::= T$	$LHS.trans = T.trans$
P2 $E ::= E \text{ plus } T$	$LHS.trans = \text{AddNode}$ where $\text{AddNode.arm1} = E.trans$ $\text{AddNode.arm2} = T.trans$
P3 $T ::= F$	$LHS.trans = F.trans$
P4 $T ::= T \text{ times } F$	$LHS.trans = \text{MultNode}$ where $\text{MultNode.arm1} = T.trans$ $\text{MultNode.arm2} = F.trans$
P5 $F ::= \text{lpar } E \text{ rpar}$	$LHS.trans = E.trans$
P6 $F ::= \text{id}$	$LHS.trans = \text{IDNode}$ where $\text{IDNode.content} = \text{id.value}$

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)



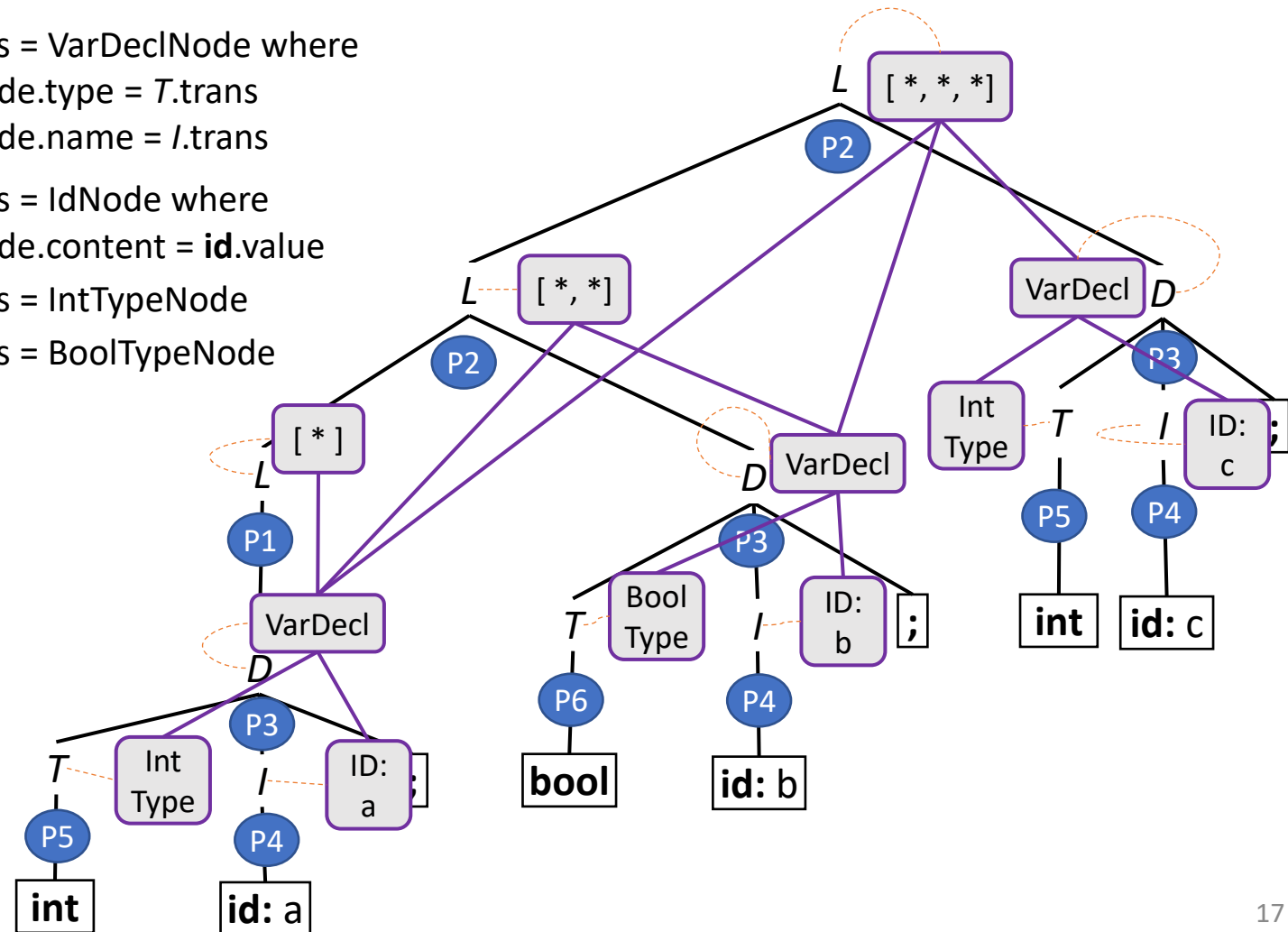
Capturing Program Constructs

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;



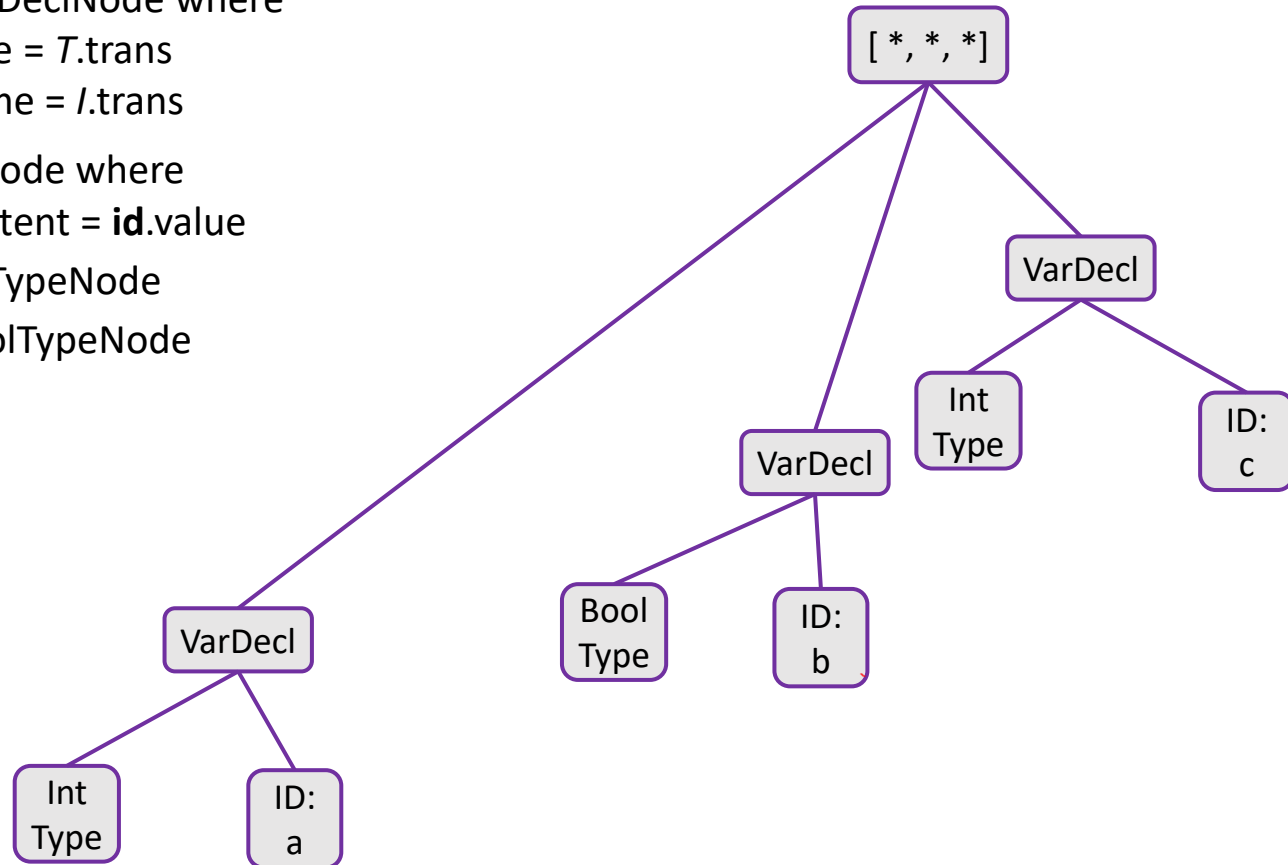
Capturing Program Constructs

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;



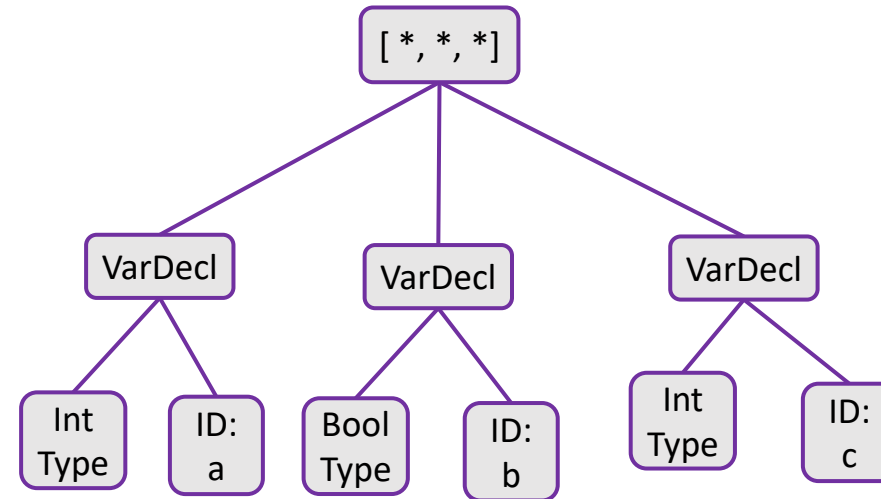
Capturing Program Constructs

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

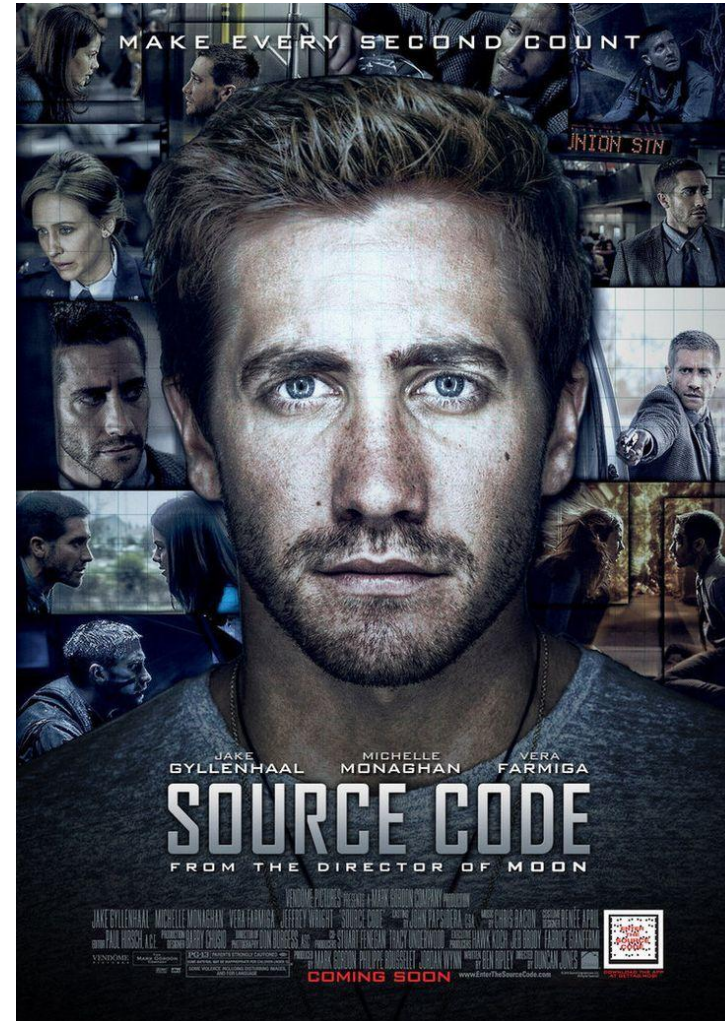


**Syntax-Directed
Definition**

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;  
};
```

CFG Production Rule

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(){ ... }
};

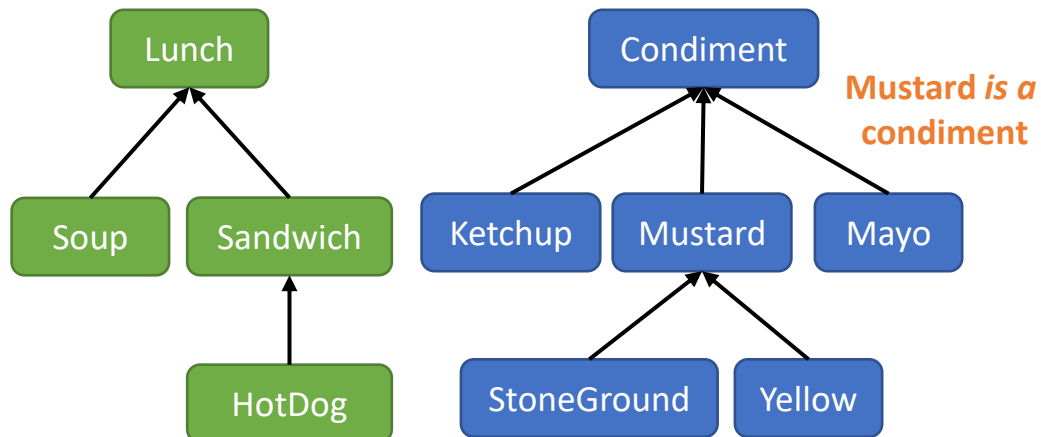
class HotDog : public Sandwich{
public:
    Ketchup * myK;
    Mustard * myM;
};
```

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

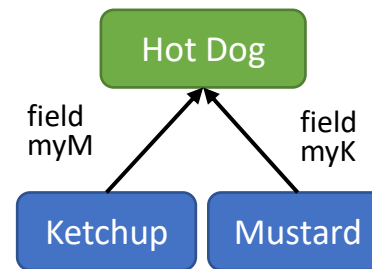
A hot dog *is a* sandwich

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

Abstract Syntax Trees – ASTs in Code

Naïve

AST Implementation

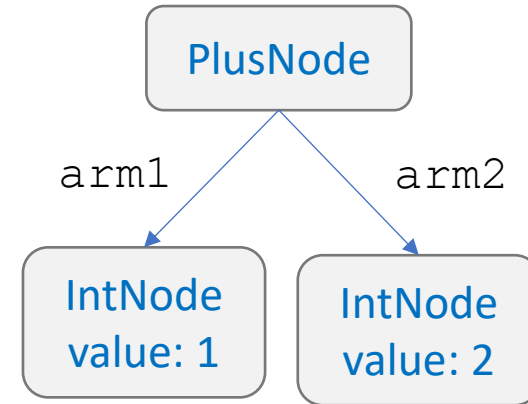
```
class PlusNode
{
    IntNode& arm1;
    IntNode& arm2;
};
```

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

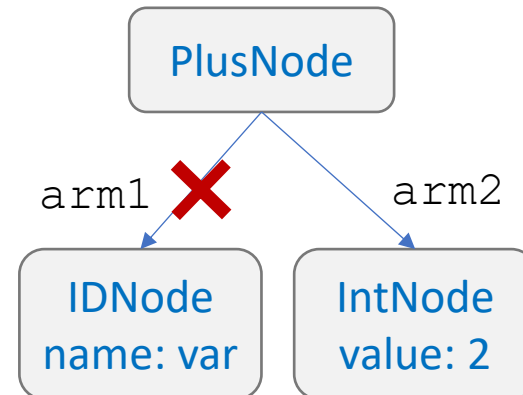
```
class IDNode
{
    std::string content; };
```

*Oops! Disallows
Variables in
expressions!*

AST for 1+2



AST for var+2



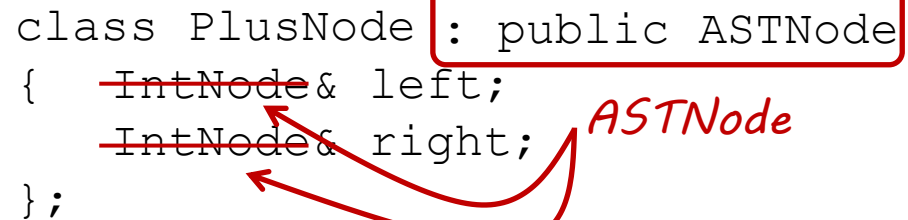
ASTs: Connecting Subtrees

Abstract Syntax Trees – ASTs in Code

Naïve AST Implementation

```
class ASTNode{ ... };
```

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



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

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

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

Class hierarchy for AST Nodes

ASTNode

└─> IntNode

└─> IDNode

└─> DeclNode

⋮

ASTs: Connecting Subtrees

Abstract Syntax Trees – ASTs in Code

Less Naïve
~~Naïve~~

AST Implementation

```
class ASTNode{ };
```

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

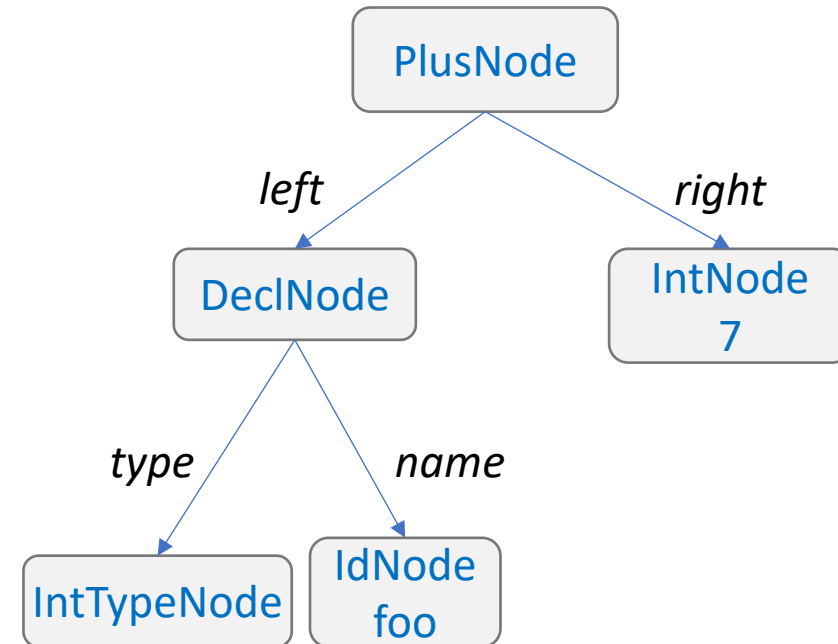
```
class IntNode : public ASTNode  
{   int value; };
```

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

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

*Wrongly allows
all node types!
Too generic!*

int foo; + 7



ASTs: Connecting Subtrees

Abstract Syntax Trees – ASTs in Code

Ok
~~Less Naïve~~
~~Naïve~~

AST Implementation

```
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

Class hierarchy for AST Nodes

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