

Compiler Techniques

4. Semantic Analysis

Huang Shell Ying

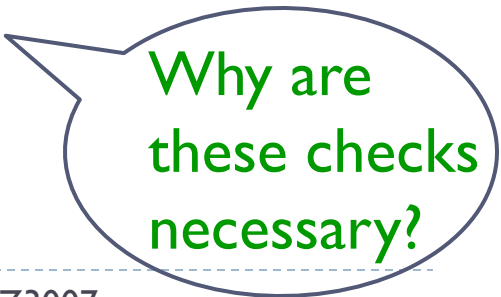
The Compiler So Far

- ▶ Lexical analysis -- Detects illegal characters
- ▶ Parsing -- Detects syntax errors
- ▶ Semantic analysis -- Catches all remaining errors
 - ▶ Last “front end” phase
- ▶ Why a separate semantic analysis?
 - ▶ The semantic rules cannot be expressed with context-free grammars
 - ▶ To make the compiler easier to understand, extend and maintain – separation of concerns

Ref: I_Introduction
Structure of a compiler

What Does Semantic Analysis Do?

- ▶ Checks whether semantic rules are violated:
 - ▶ Scoping errors
 - ▶ All variable names are declared
 - ▶ Methods/functions called are declared
 - ▶ Variable/method/class names are declared only once in a scope
 - ▶ Type inconsistencies
 - ▶ Operators and operands are type-compatible
 - ▶ Functions/methods are called with correct number of parameters and types
 - ▶ Return expression compatible with function type
 - ▶ And others ...
- ▶ The requirements depend on the language.



Why are these checks necessary?

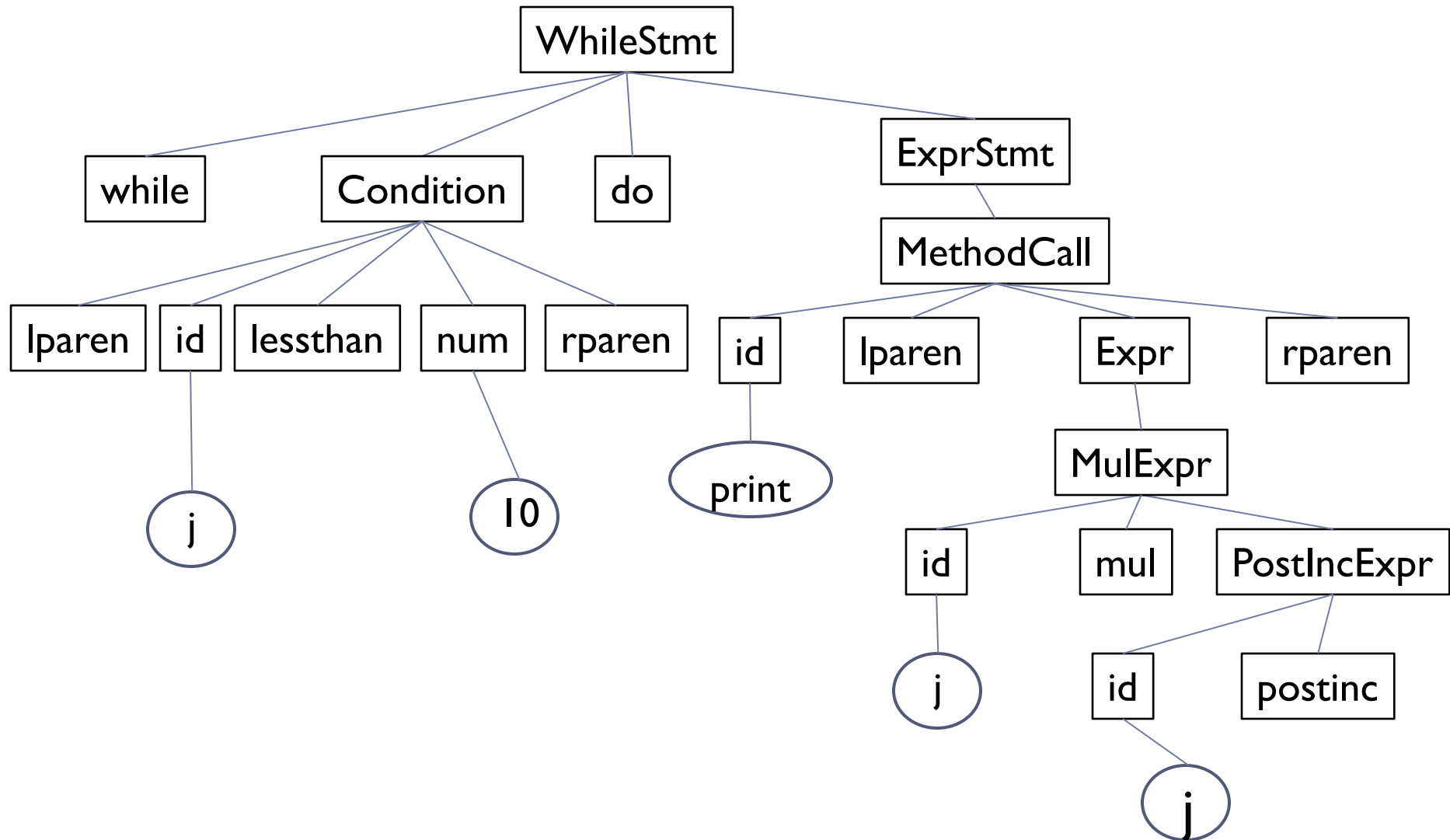
Abstract Syntax Tree (AST)

- ▶ The AST is a central data structure for all post-parsing activities like semantic checking, code generation, etc.
- ▶ The AST catches the source program structure, omitting the unnecessary syntactic details.
- ▶ The **parser** recognizes the syntactic structure of the source program and builds the AST.
- ▶ The AST should be concise, but sufficiently flexible to accommodate the diversity of post-parsing phases.

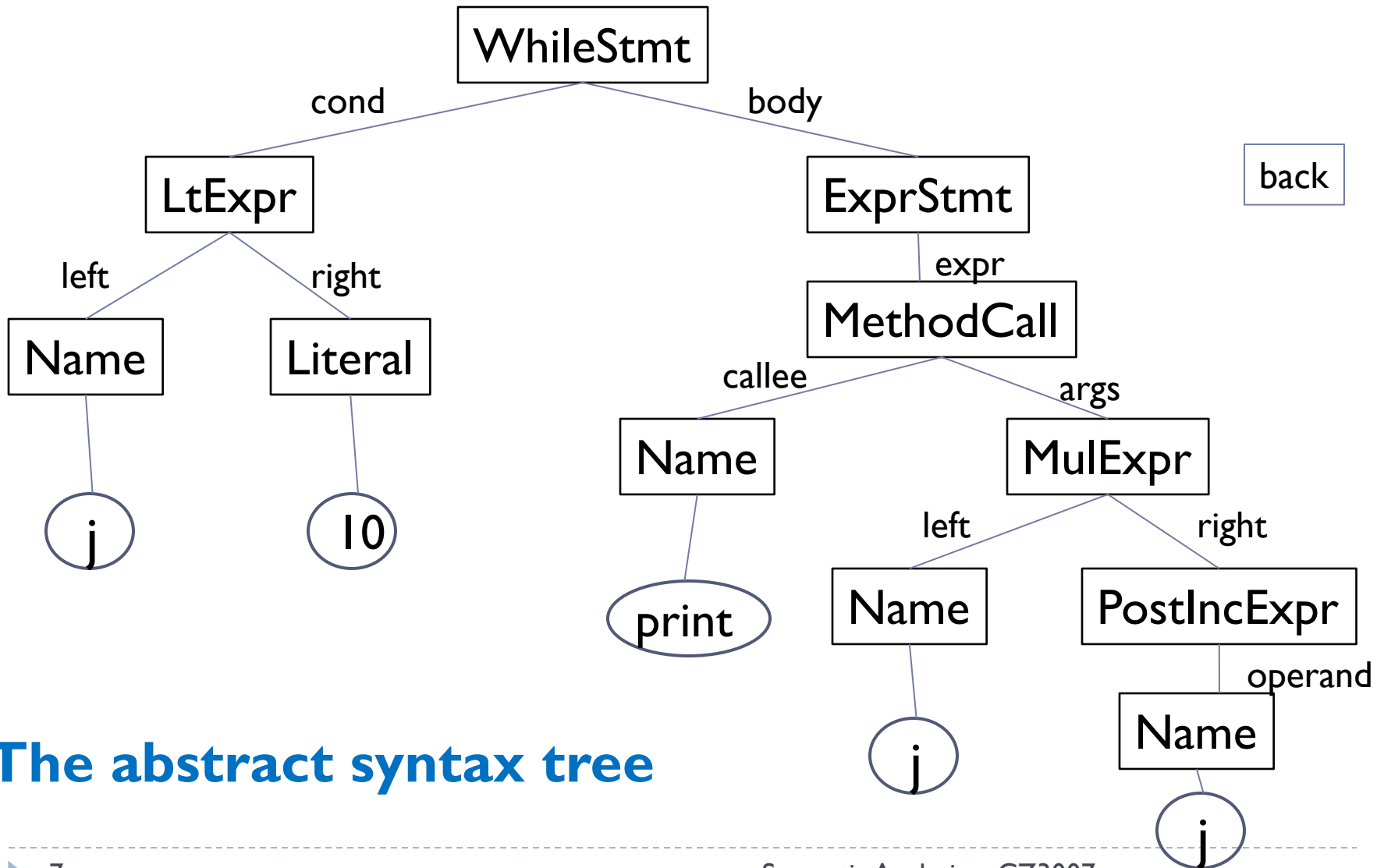
Differences between the Parse Trees & ASTs

- ▶ The role and purpose of the AST and parse trees are different:
 - ▶ A parse tree represents a derivation – a conceptual tool for analysing the parsing process. E.g., next slide.
 - ▶ An AST is a real data structure and stores information for semantic analysis and code generation. E.g., slide 7.
- ▶ The tree structure:
 - ▶ A parse tree contains a node for every token in the program.
 - ▶ Semantically useless symbols and punctuation tokens are omitted in AST.

Example of a Parse Tree



Example of an AST



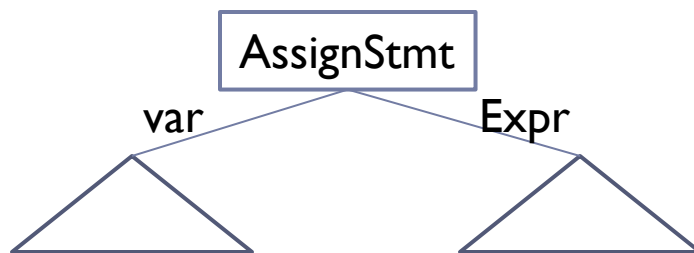
The abstract syntax tree

Design of the AST

We devise the AST for the grammar to **retain the essential information needed for name and type analysis and code generation.**

Examples

- Assignment statement: the target of the assignment and the value to be stored at that target.



AST node:

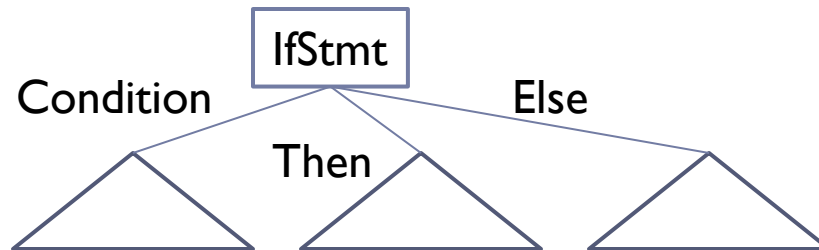


AST subtree:

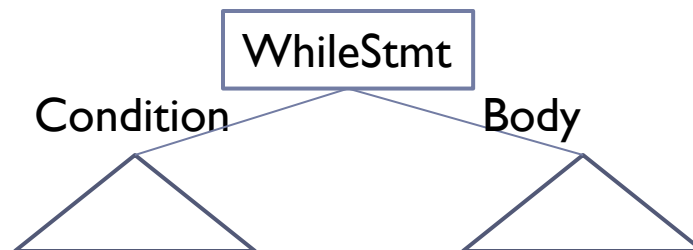


Design of the AST

- ▶ If statement: the condition to be tested and the statements to be executed in the two alternatives.



- ▶ While statement: the loop condition and the loop body.



AST

Slide 7

- ▶ Every node in the AST is labelled by a *node type*.
- ▶ Every edge in the AST is also labelled by a *child name* that expresses the *structural relationship* between the parent node and the child node.
- ▶ A node that have link(s) to child node(s) is a *nonterminal* node.
- ▶ A node with no child node will have values like strings or numbers. Such a node is a *terminal* node.

AST

- ▶ The structure of ASTs for a programming language can be implemented by a **class hierarchy** in an object-oriented language. (class hierarchy defines the inheritance relationships between classes)
E.g., AddExpr and LtExpr are subclasses of Expr.
- ▶ These classes are called *AST classes* since they model nodes in the AST.
- ▶ The structure of ASTs can be described by an **abstract grammar**.
- ▶ The translation from abstract grammars to Java classes is implemented by the **JastAdd** tool (JastAdd: *just add* to the *ast*) .

Slide 7

Abstract grammars



- ▶ **Abstract grammars** define **sets of trees** (namely, abstract syntax trees), while **context free grammars** define **sets of sentences**.

Slide 7

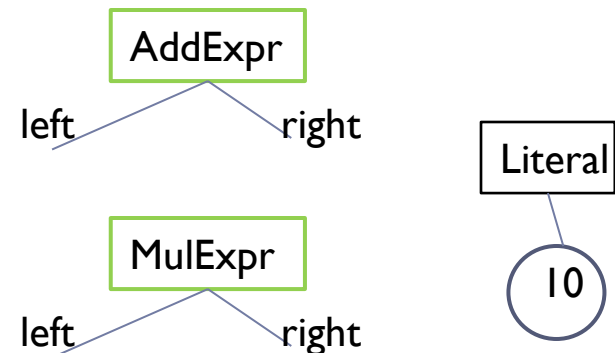
- ▶ Abstract grammars correspond to a class hierarchy.
- ▶ Abstract grammars can easily be translated into sets of Java classes, with every node type represented by one Java class.

Abstract grammars

- ▶ An abstract grammar consists of a set of rules. Each rule defines the structure of one class in the class hierarchy.
- ▶ In abstract grammars, there is only a single rule for each class.

An example of **CFG** for arithmetic expression with addition and multiplication:

Expr = Expr PLUS Term
 | Term
Term = Term MUL Factor
 | Factor
Factor = NUMBER
 | LPAREN Expr RPAREN



Example of an Abstract Grammar

The corresponding **abstract grammar**:

abstract Expr;

abstract BinExpr : Expr ::= **Left : Expr** Right : Expr;

AddExpr : BinExpr;

MulExpr : BinExpr;

Literal : Expr ::= **<Value:Integer>** ;

The LHS and the RHS are separated by **::=**, with the node type defined by the LHS of the rule.

Non-terminal child node

Expr is the type of the child node

BinExpr inherits from Expr

AddExpr inherits from BinExpr

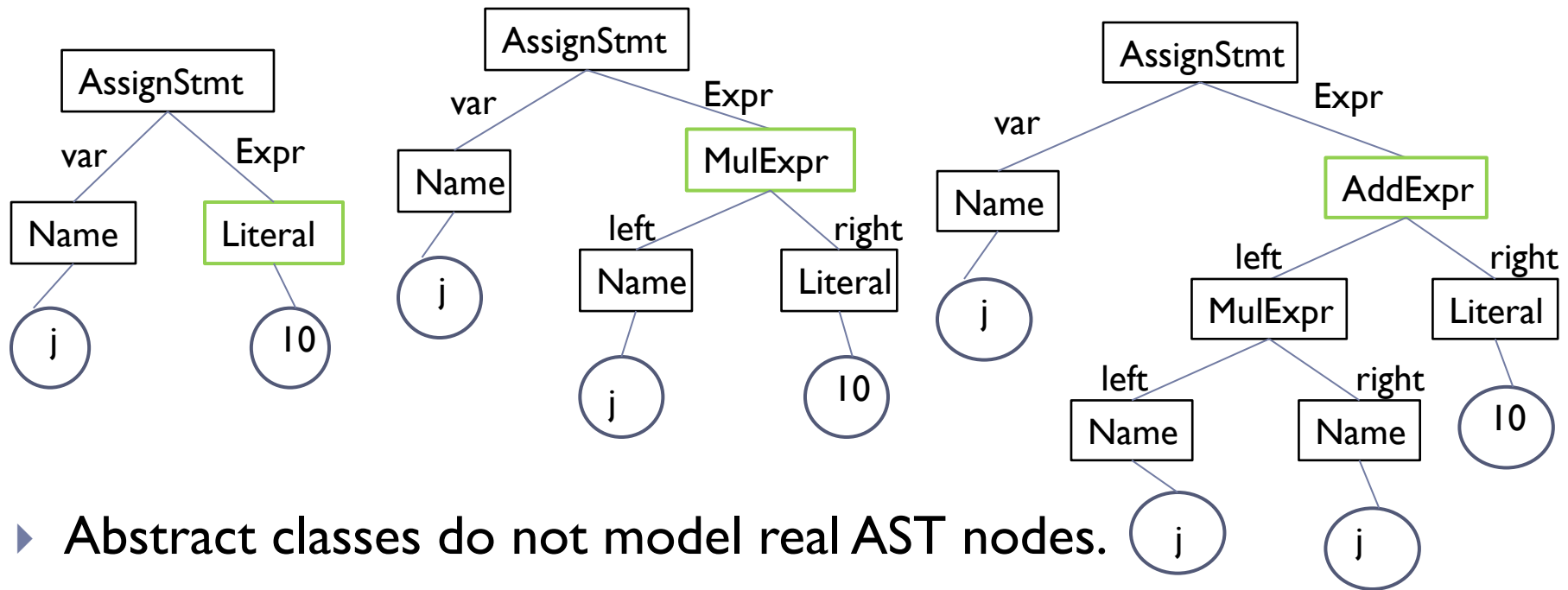
Terminal child node

back

Design of the abstract grammar

Why do we have the abstract class Expr?

Why do we have the abstract class BinExpr?



- ▶ Abstract classes do not model real AST nodes.
- ▶ Abstract classes are useful to **structure inheritance hierarchies** and to **encapsulate common structure** of their child types.

Java classes derived from the abstract grammar in Slide 14

```
// abstract Expr;
```

```
public abstract class Expr {}
```

Abstract node types give rise to abstract classes

Every class gets a constructor that creates a new AST node

```
// abstract BinExpr : Expr ::= Left:Expr Right:Expr;
```

```
public abstract class BinExpr extends Expr {
```

```
    private final Expr Left, Right;
```

```
    public BinExpr(Expr Left, Expr Right) {
```

```
        this.Left = Left;    this.Right = Right;
```

```
    }
```

```
    public Expr getLeft() { return Left; }
```

```
    public Expr getRight() { return Right; }
```

```
}
```

Children of a node type become fields of the Java class

Back to slide 25

Every class gets the getter methods for accessing the fields

back

Java classes derived from the abstract grammar in Slide 14

// AddExpr : BinExpr;

back

```
public class AddExpr extends BinExpr {  
    public AddExpr(Expr Left, Expr Right) {  
        super(Left, Right);  
    }  
}
```

Inheritance between node types becomes inheritance between the corresponding classes

// MulExpr : BinExpr;

```
public class MulExpr extends BinExpr {  
    public MulExpr(Expr Left, Expr Right) {  
        super(Left, Right);  
    }  
}
```

Java classes derived from the abstract grammar in Slide 14

```
// Literal : Expr ::= <Value:Integer>;  
public class Literal extends Expr {  
    private final Integer Value;  
    public Literal(Integer Value) {  
        this.Value = Value;  
    }  
    public Integer getValue() { return Value; }  
}
```

[back](#)

[Back to slide 76](#)

Optional Children Nodes

- ▶ Often, some node type in ASTs has a child that may or may not be present. E.g., node type `IfStmt`.
- ▶ In an abstract grammar, we can express this by bracketing the child. E.g.,

`IfStmt : Stmt ::= Cond:Expr Then:Stmt [Else:Stmt];`

[back](#)

- ▶ The `Opt` class is used by `JastAdd` to encode optional nonterminal children:

Optional Children Nodes

```
public class Opt<T> {  
    private final T child;  
    // child is present  
    public Opt(T child) {  
        this.child = child;  
    }  
    // child is absent  
    public Opt() {  
        this.child = null;  
    }  
    public T get() { return child; }  
}
```

Optional Children Nodes

- ▶ Part of the class implementing an IfStmt nonterminal with optional else branch:

Slide 19

```
public class IfStmt extends Stmt {  
    private final Expr Cond;  
    private final Stmt Then;  
    private final Opt<Stmt> Else;  
    // constructors omitted  
    // getCond, getThen omitted  
    // check for presence of else branch  
    public boolean hasElse() {  
        return Else.get() != null;  
    }  
    public Stmt getElse() {  
        return Else.get();  
    }  
}
```

List Children Nodes

- ▶ Many nonterminals also may have an arbitrary number of children of the same type. E.g., BlockStmt.
- ▶ In an abstract grammar, this is encoded as a **list** child, which looks like a normal nonterminal child, but is suffixed with a *. For example,

BlockStmt : Stmt ::= Stmt*;

- ▶ This declaration says that a block statement can have an arbitrary number (including zero!) of Stmt child nodes.

Java classes derived from
 $\text{BlockStmt} : \text{Stmt} ::= \text{Stmt}^*;$

```
public class BlockStmt extends Stmt {  
    private final List<Stmt> stmts;  
    public BlockStmt(List<Stmt> stmts) {  
        this.stmts = stmts;  
    }  
  
    public List<Stmt> getStmts() { return stmts; }  
    public Stmt getStmt(int i) {  
        return stmts.get(i);  
    }  
  
    public int getNumStmt() {  
        return stmts.size();  
    }  
}
```

[back](#)

Slide 12

Generating ASTs

- ▶ Once the Java classes for all AST node types have been generated using JastAdd, semantic actions can be placed in the grammar (CFG) to extend the parser to construct the AST.
- ▶ The actions construct AST nodes by using the Java classes and constructors generated by JastAdd.
- ▶ The actions themselves are surrounded by `{:` and `;}` and have to be placed at the very end of a rule.
- ▶ Nonterminals appearing on the right hand side of a CFG rule can be named. These names can be used in the action to refer to the values computed when reducing these nonterminals.

Slide 12

Semantic actions added to a Beaver specification (Example)

Slide 13

```
%terminals PLUS, MUL, NUMBER, LPAREN, RPAREN;
```

```
%goal Expr;
```

```
%typeof NUMBER = "String";
```

The **%typeof** declares Java types for grammar symbols

```
%typeof Expr, Term, Factor = "Expr";
```

```
Expr = Expr.e PLUS Term.t { return new AddExpr(e, t); : }
```

Slide 17

```
  | Term.t { return t; : }
```

E.g. the Term occurrence on the right hand side is named t

```
  ;
```

```
Term = Term.t MUL Factor.f { return new MulExpr(t, f); : }
```

```
  | Factor.f { return f; : }
```

```
  ;
```

Slide 18

```
Factor = NUMBER.n { return new Literal(Integer.parseInt(n)); : }
```

```
  | LPAREN Expr.e RPAREN { return e; : }
```

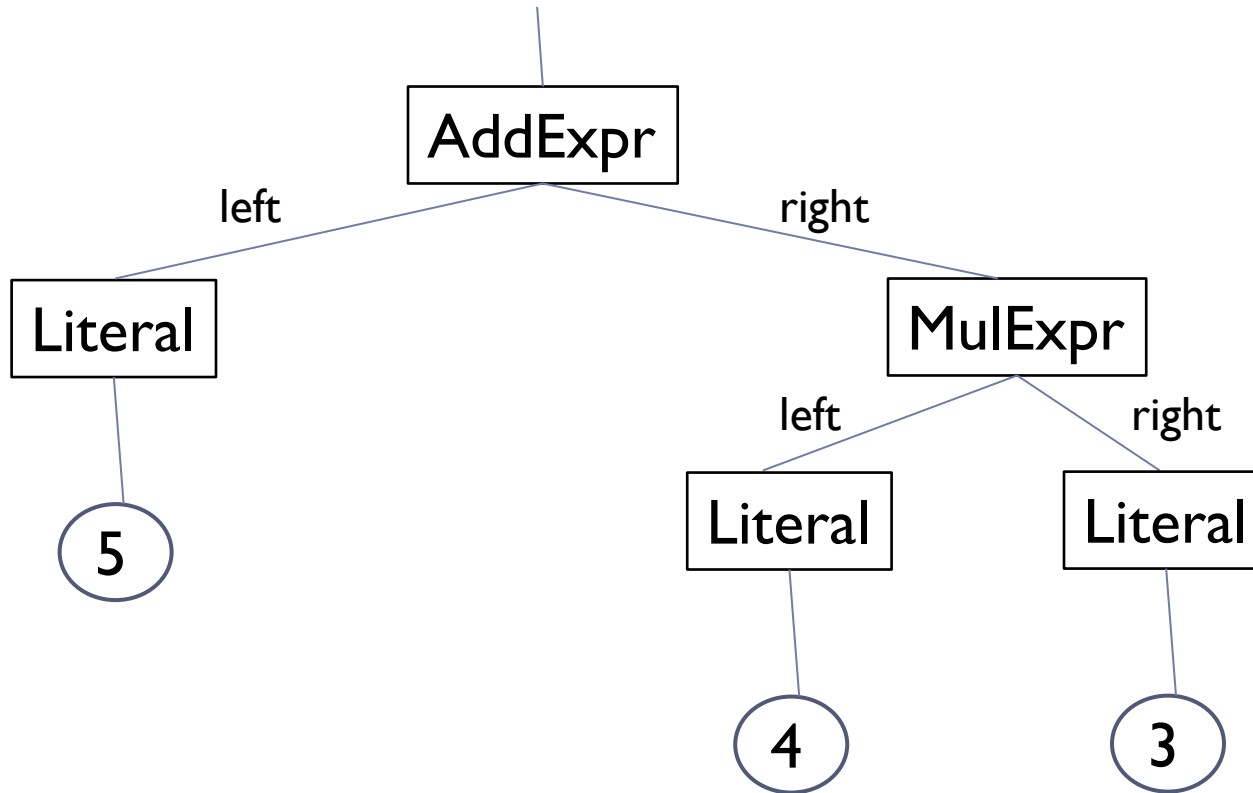
Slide 14

```
  ;
```

The Parser generates the AST

- ▶ Given this specification, Beaver builds a LALR(1) parse table.
- ▶ The generated parser uses the parse table to process the input tokens as usual.
- ▶ Every time the parser performs a **reduce()** action it executes the action associated with that rule – an AST node is built.
- ▶ When the start nonterminal is reduced, the whole AST is built.

Example of the AST for “5+4*3”

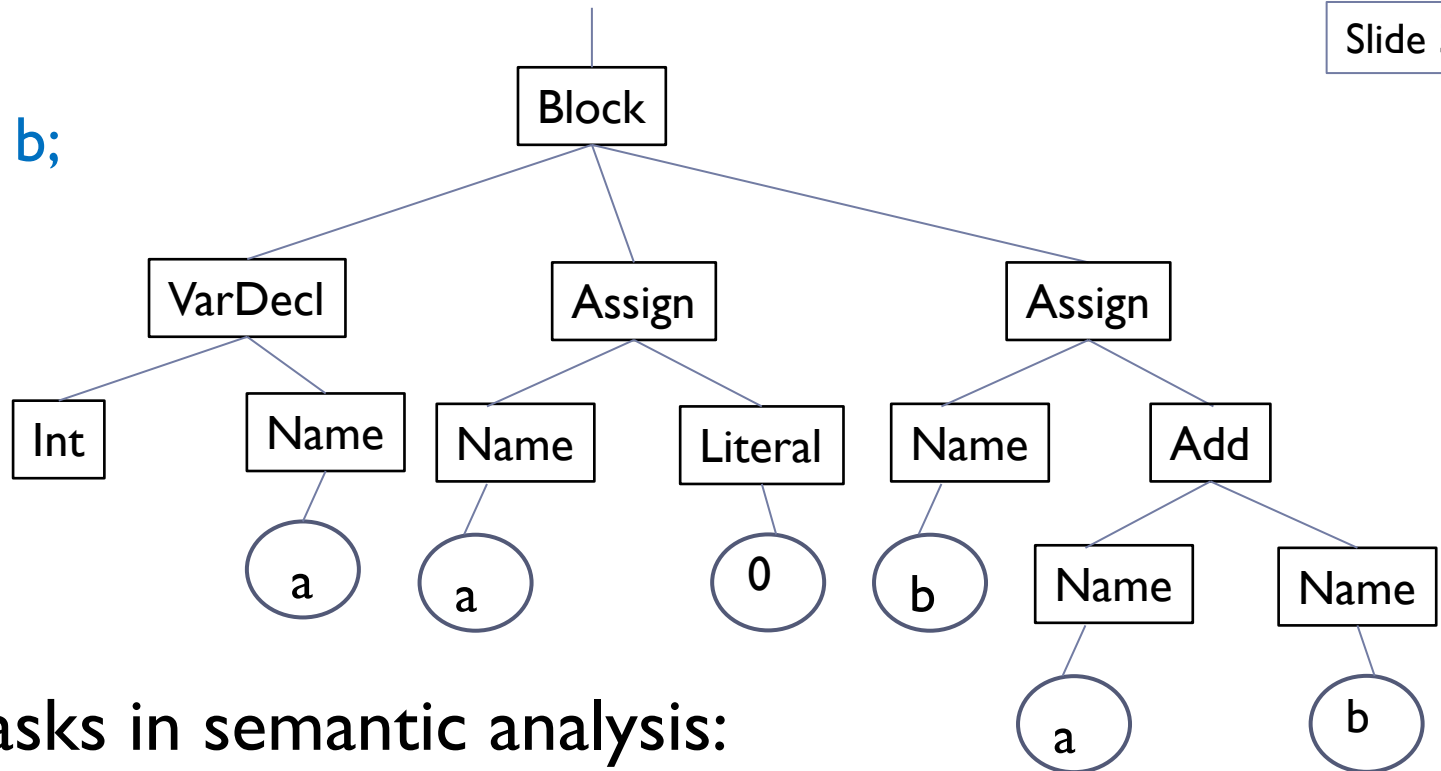


Semantic analysis using ASTs

- ▶ Example: a simple program has the AST shown

```
{ int a;  
  a = 0;  
  b = a + b;  
}
```

Slide 3



- ▶ Two tasks in semantic analysis:
 - 1) name checking based on name analysis
 - 2) type checking based on type analysis

Performing Computations on ASTs

- ▶ We will need to perform computations on the AST of a program.
- ▶ AST nodes need to have additional methods to compute information like where its declaration is or what its type is.
- ▶ Possible ways for the compiler builder to define the methods on the AST nodes
 1. **Internal definitions:** insert methods into AST classes which are automatically generated by JastAdd
 2. **External definitions:** collect all methods for semantic checking into a separate class.
 3. **The visitor pattern:** a variant of the external definition approach.

Slide 17

Inter-type methods

- ▶ Inter-type methods is an alternative to visitors.
- ▶ **JastAdd** offers a language feature called **inter-type methods** that is not available in Java.
- ▶ An inter-type method is a method of an AST class, but it is defined independent from the class.
- ▶ Several such inter-type methods that logically belong together can be collected together into an **aspect** and placed in a file with the extension **.jadd**.
- ▶ However, Java does not know about aspects and inter-type methods.

Inter-type methods

- ▶ But when JastAdd generates the Java classes corresponding to node types, it can simply insert the definitions of the inter-type methods into the right AST class.
- ▶ Inter-type declarations combine the advantages of internal and external definitions.
- ▶ The only disadvantage of inter-type declarations is that they are not a Java feature, but have to be translated into normal Java methods by JastAdd.

Slide 17

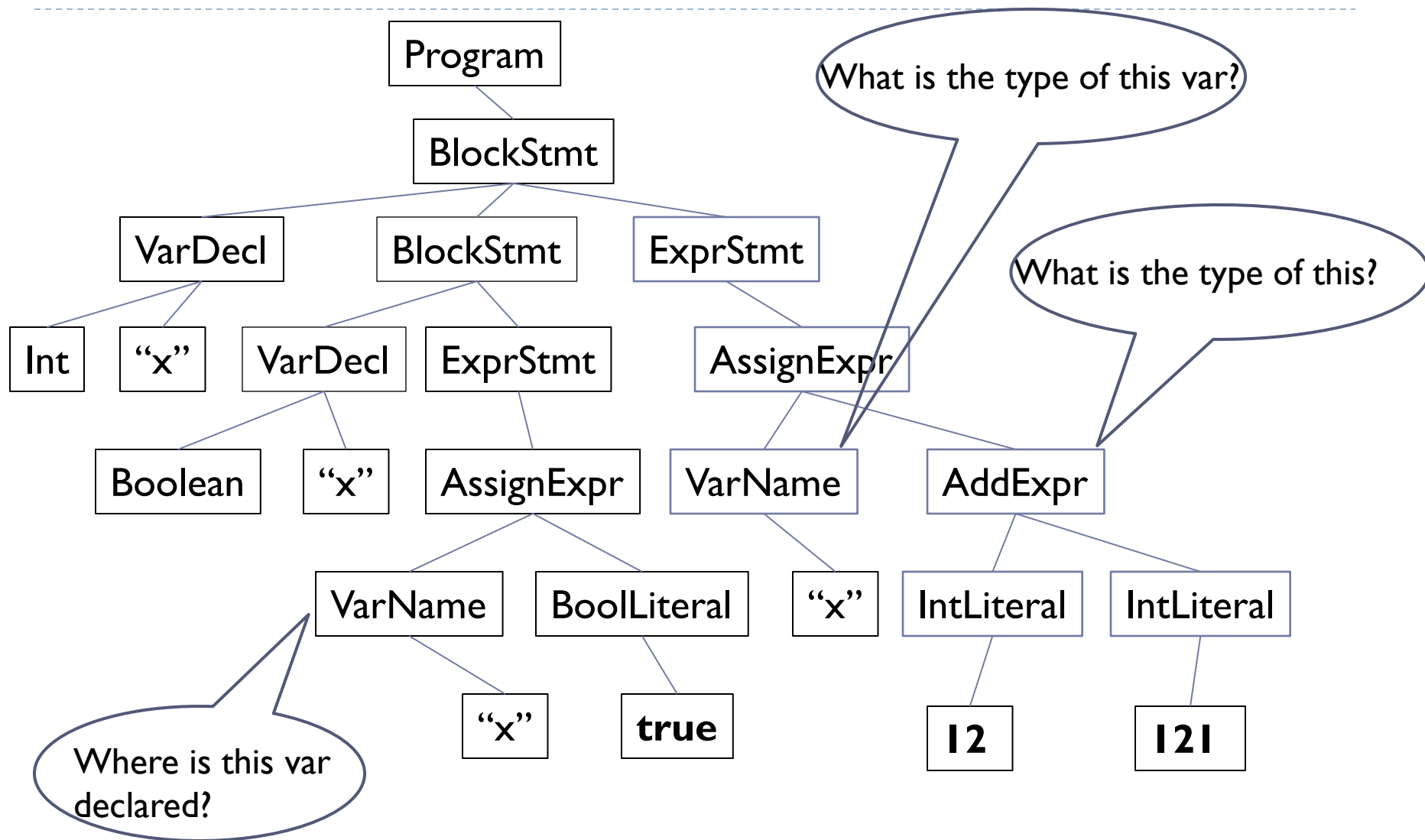


Attribute Grammars

- ▶ Many computations on ASTs follow common patterns:
 1. Many analyses need to compute some value for an AST node based on values that were already computed for this node or for the node's children.
 2. Another pattern is that the value computed for a given node depends on values computed for the node's parent node, or even its sibling nodes.

For example:

Attribute Grammars



Attribute Grammars

- ▶ Computations following these commonly occurring patterns can be described by **attribute grammars**.
- ▶ An attribute grammar defines a set of attributes on top of an abstract grammar.
- ▶ An attribute is simply a function that computes a value for a certain kind of nodes.

Slide 14

Running Example

An abstract grammar for describing box scenarios like the one in next slide:

Frame ::= Box;

abstract Box;

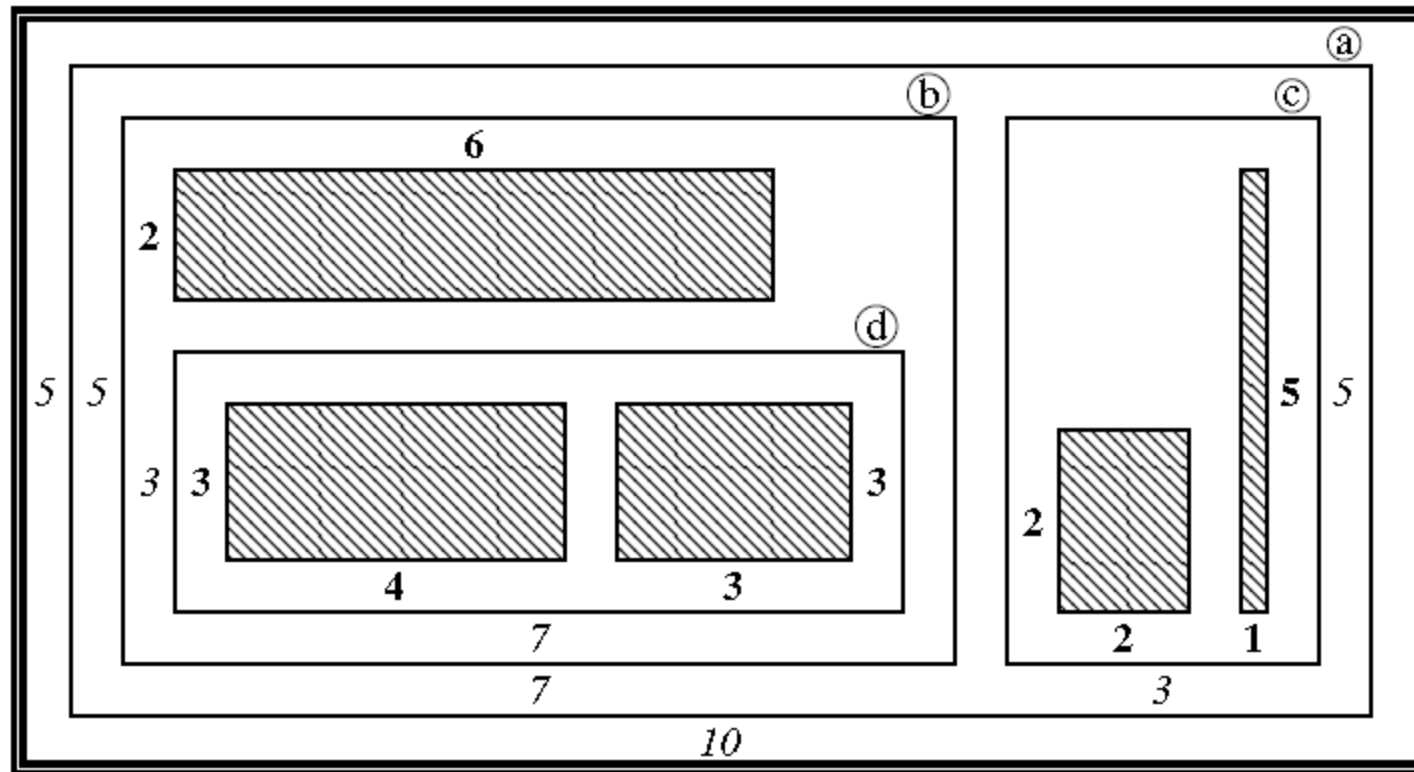
HBox : Box ::= Left:Box Right:Box;

VBox : Box ::= Top:Box Bottom:Box;

ABox : Box ::= <Width:Integer> <Height:Integer>;

back

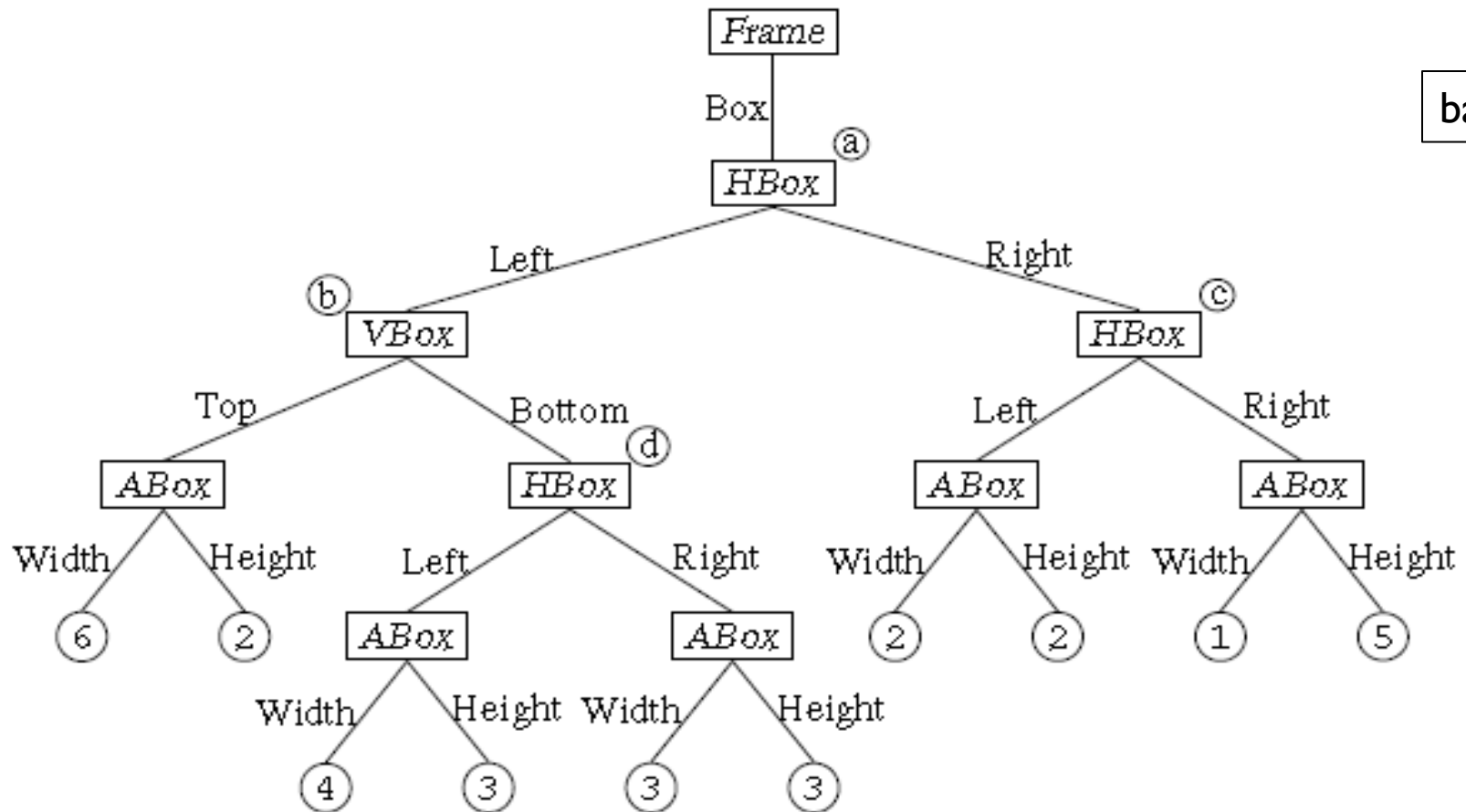
An example of a box scenario



back

An example box scenario: atomic boxes are shaded, bold numbers indicate width and height of atomic boxes, italic numbers width and height of containing boxes (dimensions are not depicted accurately)

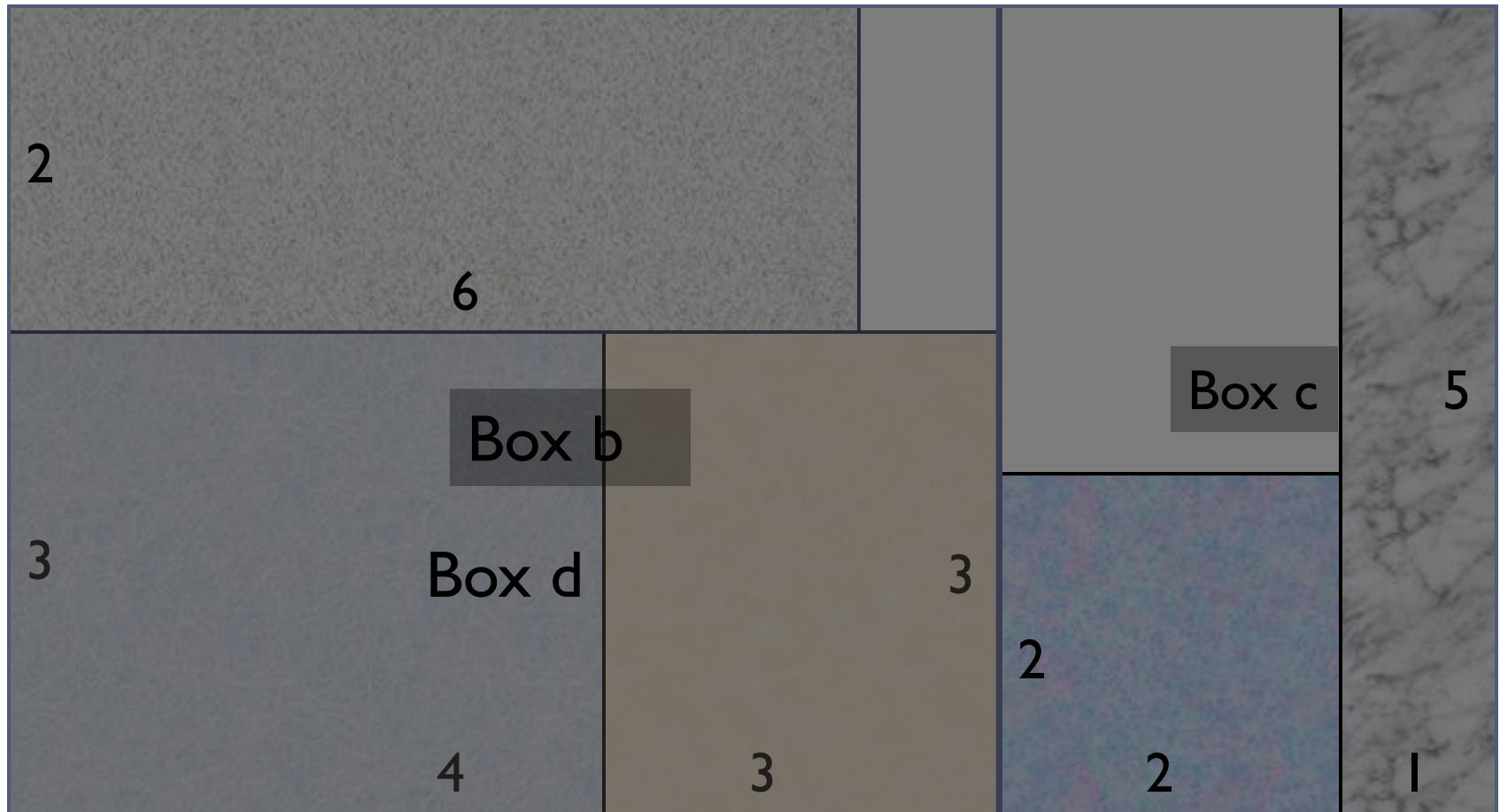
The AST representation of the scenario



Synthesised attributes

- ▶ A **synthesised attribute** a is an attribute whose value on some node n is computed based on the attribute values of the **children** of n and/or some other attributes of n .
- ▶ In our abstract grammar, only atomic boxes have explicit widths and heights.
- ▶ The widths and heights of horizontal and vertical boxes are examples of **synthesised attributes**.

back



What are the computations of the width and height of a box based on?

Defining synthesised attributes

Slide 34

width is a synthesised attribute of AST node Box

Attribute type

This attribute does not take any arguments

syn int Box.width();

eq VBox.width() = Math.max(getTop().width(),
getBottom().width());

eq HBox.width() = getLeft().width() + getRight().width();

eq ABox.width() = getWidth();

an equation that defines the attribute for a particular node type

The RHS of an equation can be any Java expression

- ▶ A synthesised attribute must be defined for every node type on which it is declared. However, node types inherit attribute definitions from their super types.

Java methods generated for the synthesised attribute width

```
public abstract class Box {
```

```
...
```

```
abstract int width();
```

```
}
```

```
public class VBox extends Box {
```

```
...
```

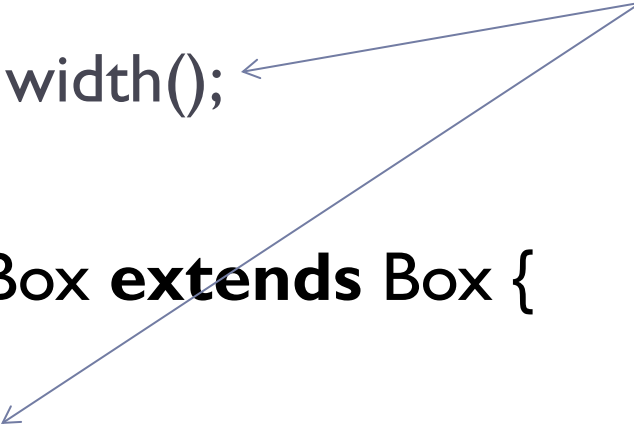
```
int width() {
```

```
    return Math.max(getTop().width(),  
                    getBottom().width());
```

```
}
```

```
}
```

JastAdd adds these
methods into the
appropriate classes



Java methods generated for the synthesised attribute width

```
public class HBox extends Box {
```

```
...
```

```
int width() {
```

```
    return getLeft().width() + getRight().width();
```

```
}
```

```
}
```

```
public class ABox extends Box {
```

```
...
```

```
int width() {
```

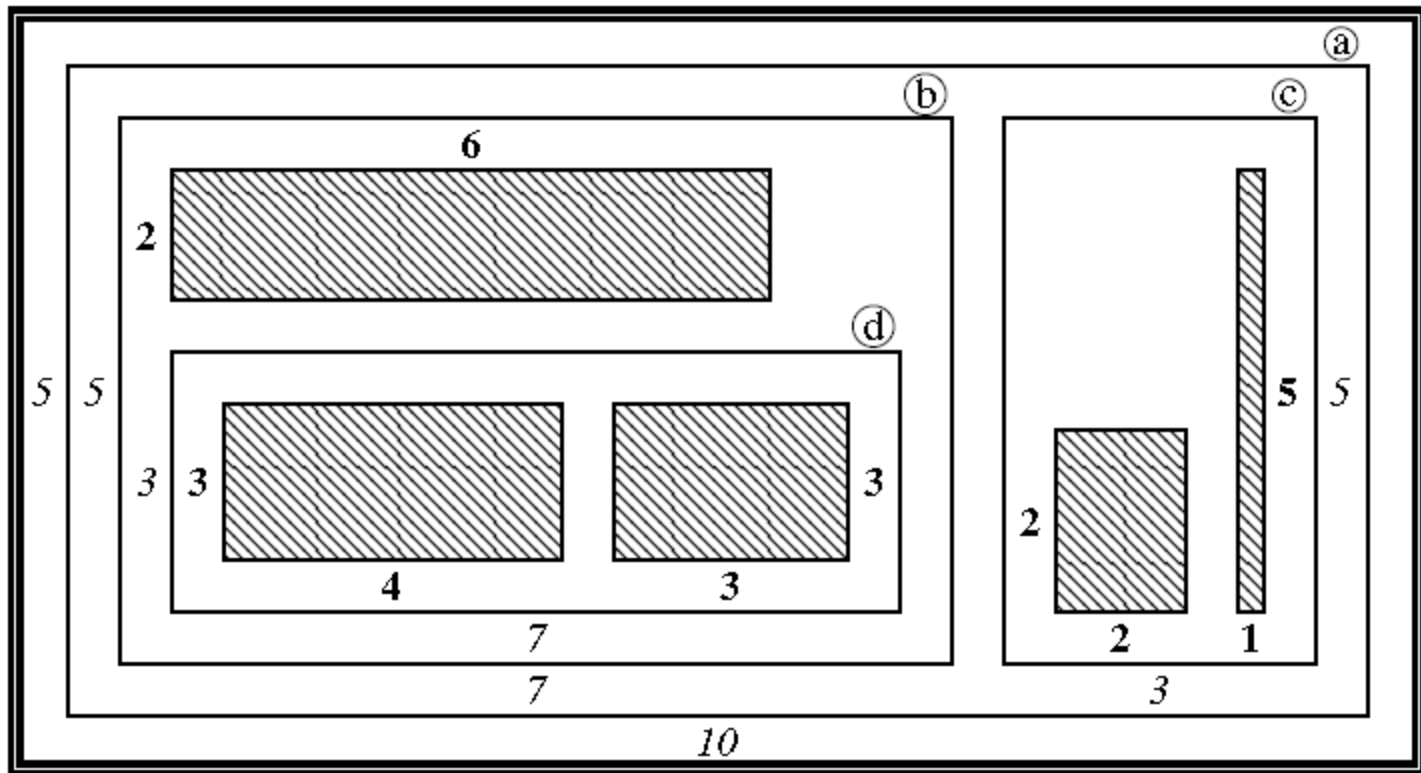
```
    return getWidth();
```

```
}
```

```
}
```

Inherited attributes

- Suppose we additionally want to find out the coordinates of every box's lower left corner.

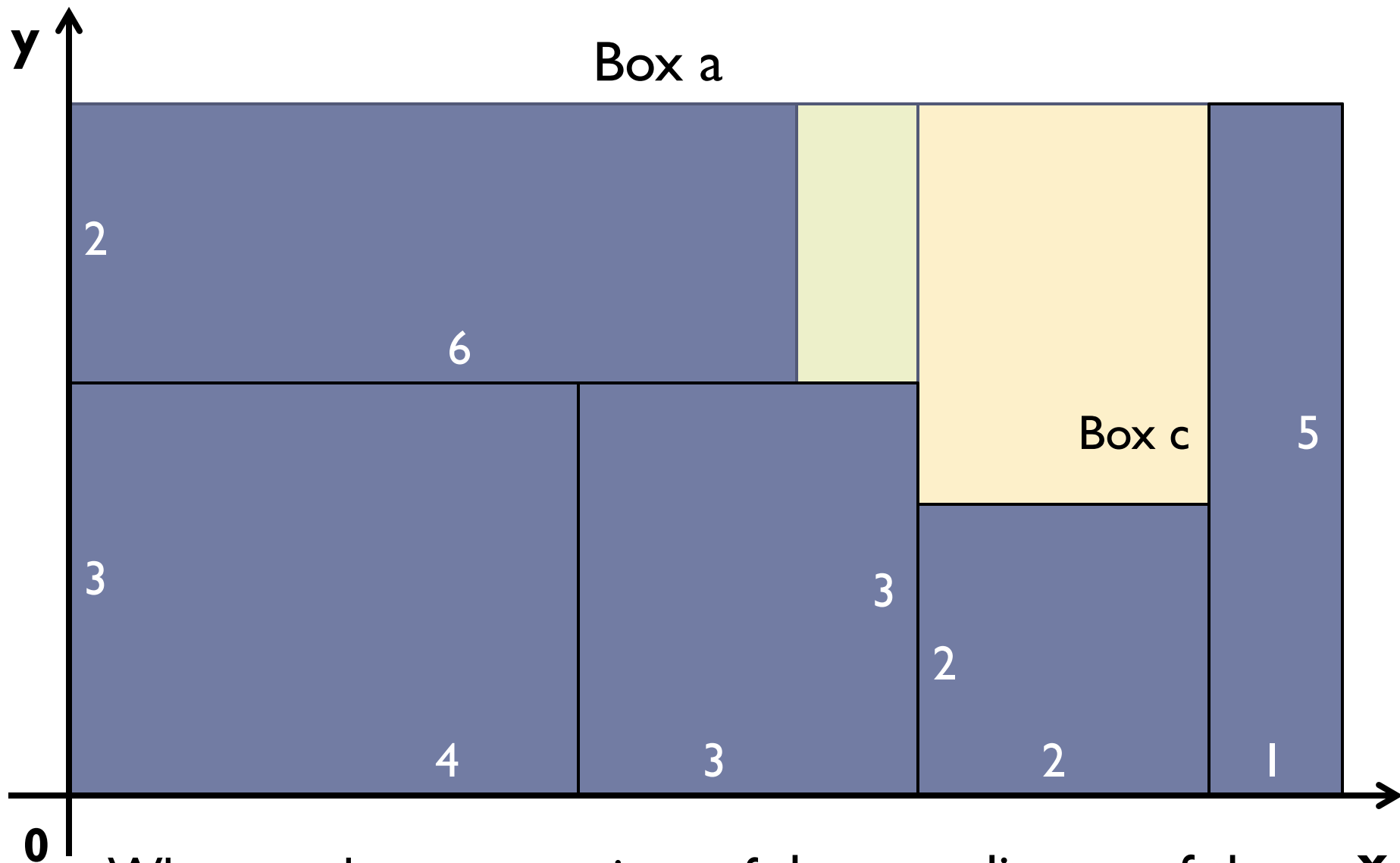


Assume the Frame's lower left corner is at (0,0)

back

Inherited attributes

- ▶ The x and y coordinate values of the lower left corner are the attributes of concern.
- ▶ They are examples of *inherited attributes*.
- ▶ A *inherited attribute* **a** is an attribute whose value on some node **n** is computed based on the attribute values on its *parent* node and its *siblings*.



What are the computations of the coordinates of the lower left corner of a box based on?

Defining Inherited attributes (x coordinates)

xpos is an inherited attribute of AST node Box

Provide one equation for every occurrence of a relevant node type on the right side of a grammar rule for HBox

inh int Box.xpos();

eq Frame.getBox().xpos() = 0;

// Equation 1

eq HBox.getLeft().xpos() = xpos();

// Equation 2

eq HBox.getRight().xpos() = xpos() + getLeft().width();

// Equation 3

eq VBox.getTop().xpos() = xpos();

// Equation 4

eq VBox.getBottom().xpos() = xpos();

// Equation 5

Parent's attribute value

sibling's attribute value

The Java code on the RHS executes in the context of the parent node

Defining inherited attributes

- ▶ JastAdd supports several abbreviations to make the definition of inherited attributes less verbose.
- ▶ If the equations for all children of a certain node type are the same, we can simply define the equation on getChild().

back

So Equations 4 and 5 above can be expressed as a single equation:

eq VBox.getChild().xpos() = xpos();

- ▶ This kind of equations are what is known as **copy equations** where **the value of the inherited attribute on the child is simply defined to be the same as its value on the parent** -- JastAdd allows to simply omit them.

Defining inherited attributes

For example, omitting the copy equations for ypos:

```
inh int Box.ypos();
```

```
eq Frame.getBox().ypos() = 0;
```

```
eq VBox.getTop().ypos() = ypos() + getBottom().height();
```

- ▶ Like synthesised attributes, inherited attributes are translated by JastAdd into Java methods on the generated AST classes.
- ▶ Both kinds of attributes are defined in attribute definition files in an **aspect** with the extension **.jrag**

Lazy attributes

- ▶ If we want to compute attributes for different nodes in the tree, we may end up computing the same attribute for the same node over and over again.

E.g. computing the width of box a then box b

Slide 36

- ▶ If we declare an attribute to be **lazy**, JastAdd will create and maintain a **cached field** automatically.
- ▶ Such an attribute value will be computed once and the result will be stored for future retrieval.

E.g. **syn lazy int Box.width();**
inh lazy int Box.xpos();

Reference attributes

- ▶ A **reference attribute** is an attribute whose value is a **reference to some other node** in the AST.

E.g., an attribute leftNeighbour that returns the box immediately to the left of a given box

Returns a
reference to the
left neighbour box

inh lazy Box Box.leftNeighbour();
eq HBox.getRight().leftNeighbour() = getLeft();
eq Frame.getBox().leftNeighbour() = **null**;

(we omit copy equations: e.g. the left neighbour of the left child of a HBox is the left neighbour of the parent box)

Slide 42

Parameterised attributes

- ▶ The attributes we have seen so far has a single value, e.g. width, xpos of node Box.
- ▶ Some attribute value of an AST node depends on some parameters.
- ▶ A parameterized attribute will have one value for each possible combination of parameter values.



Name and Type Analysis

- ▶ The goal of **name analysis** is to determine, for every identifier appearing in the program, which declaration it refers to.
- ▶ **Type analysis** computes the types of composite expressions from the types of their components.
- ▶ Based on name and type analysis, we can perform semantic error checking. For example, for Java, no two fields in the same class should have the same name.

Example language

- ▶ To make the discussion more concrete, we will show how to actually implement name and type analysis for a very simple language. Its abstract grammar:

Program ::= BlockStmt;

[back](#)

abstract Stmt;

VarDecl : Stmt ::= TypeName <Name:String>;

BlockStmt : Stmt ::= Stmt*;

IfStmt : Stmt ::= Cond:Expr Then:Stmt [Else:Stmt];

WhileStmt : Stmt ::= Cond:Expr Body:Stmt;

ExprStmt : Stmt ::= Expr;

Example language

abstract Expr;

IntLiteral : Expr ::= <Value:Integer>;

BoolLiteral : Expr ::= <Value:Boolean>;

ArrayLiteral : Expr ::= Expr*;

abstract LHSEExpr : Expr;

VarName : LHSEExpr ::= <Name:String>;

ArrayIndex : LHSEExpr ::= Base:Expr Index:Expr;

AssignExpr : Expr ::= Left:LHSEExpr Right:Expr;

abstract BinaryExpr : Expr ::= Left:Expr Right:Expr;

AddExpr : BinaryExpr;

MulExpr : BinaryExpr;

back

Example language

abstract **TypeName**;

Int : **TypeName**;

Boolean : **TypeName**;

ArrayType **TypeName** : **TypeName** ::= **ElementType** : **TypeName**;

[back](#)

- Note **ArrayLiteral** represents array literals like

[[1, 2], [3, 4], [5]]

Since the elements of an array literal may be arbitrary expressions, they can in particular themselves be array literals, and do not all have to be of the same length.

Example program

```
{  int x;
   int [][] y;
   y = [ [ 1, 2 ], [ 3 ] ];
   { // nested block
       boolean x;
       x = true;
       if (x) {
           y[0][1] = 23;
       } else {
           y[0][1] = 42;
       }
   }
}
```

[back](#)

Scoping and namespaces

- ▶ Like most languages, our language employs a scoping discipline for variables.
- ▶ The scope of a variable is the block of statements in which it is declared, including all nested blocks; this is known as *block scoping*.
- ▶ We allow scope nesting: a reference always refers to the innermost variable of that name.
- ▶ In our language, variables are the only named entities.
- ▶ In some language like Java, there are five categories of program entities that have names: packages, types (i.e., classes and interfaces), methods, variables (including fields), and labelled statements.

Scoping and namespaces

- ▶ Scopes for packages, types, methods, variables and labelled statements do not interact and can freely **overlap**.
- ▶ This is often expressed by saying that there are **separate namespaces** for these five kinds of entities.
- ▶ Traditionally, name analysis in many compilers is implemented by means of a **symbol table**.
- ▶ A symbol table is a dictionary-like data structure that associates variable names with information about the declaration they refer to.
- ▶ We will discuss a different approach.

Slide 55

Basic name analysis

- ▶ We will define an **attribute decl on node type** VarName that returns a **reference** to the VarDecl node that a given variable name refers to.
- ▶ Or returns **null** if there is no variable of that name in scope.
- ▶ Example used in this basic name analysis:

```
{ int x;  
  { boolean x;  
    x = true;  
  }  
  x = 0;  
}
```

[illegible]

Implement the decl attribute in JastAdd

- ▶ To compute the attribute **decl** for a VarName, we need to look up the place where it is defined. That is, look up the parent nodes through the AST tree.
- ▶ We define another attribute **lookupVar** for all **Expr** and **Stmt** node types of the AST tree. This attribute is an **inherited** attribute. Why?
- ▶ So to compute attribute decl of a VarName, we compute the lookupVar attribute of the Varname. The **decl** attribute is a **synthesised** attribute.

syn VarDecl VarName.decl() = lookupVar(getName());

inh lazy VarDecl Expr.lookupVar(String name);

inh lazy VarDecl Stmt.lookupVar(String name);

eq Program.getChild().lookupVar(String name) = null;

eq BlockStmt.getStmt(int i).lookupVar(String name) {
 for (int j = 0; j < i; ++j) {

 Stmt stmt = this.getStmt(j);

 if (stmt instanceof VarDecl) {

 VarDecl decl = (VarDecl) stmt;

 if (decl.getName().equals(name))

 return decl;

 }

 }

return this.lookupVar(name);

Recall copy
equations:
Slide 46

List children
nodes: Slide
23

Remember
that the
code
executes in
the context
of the
parent node

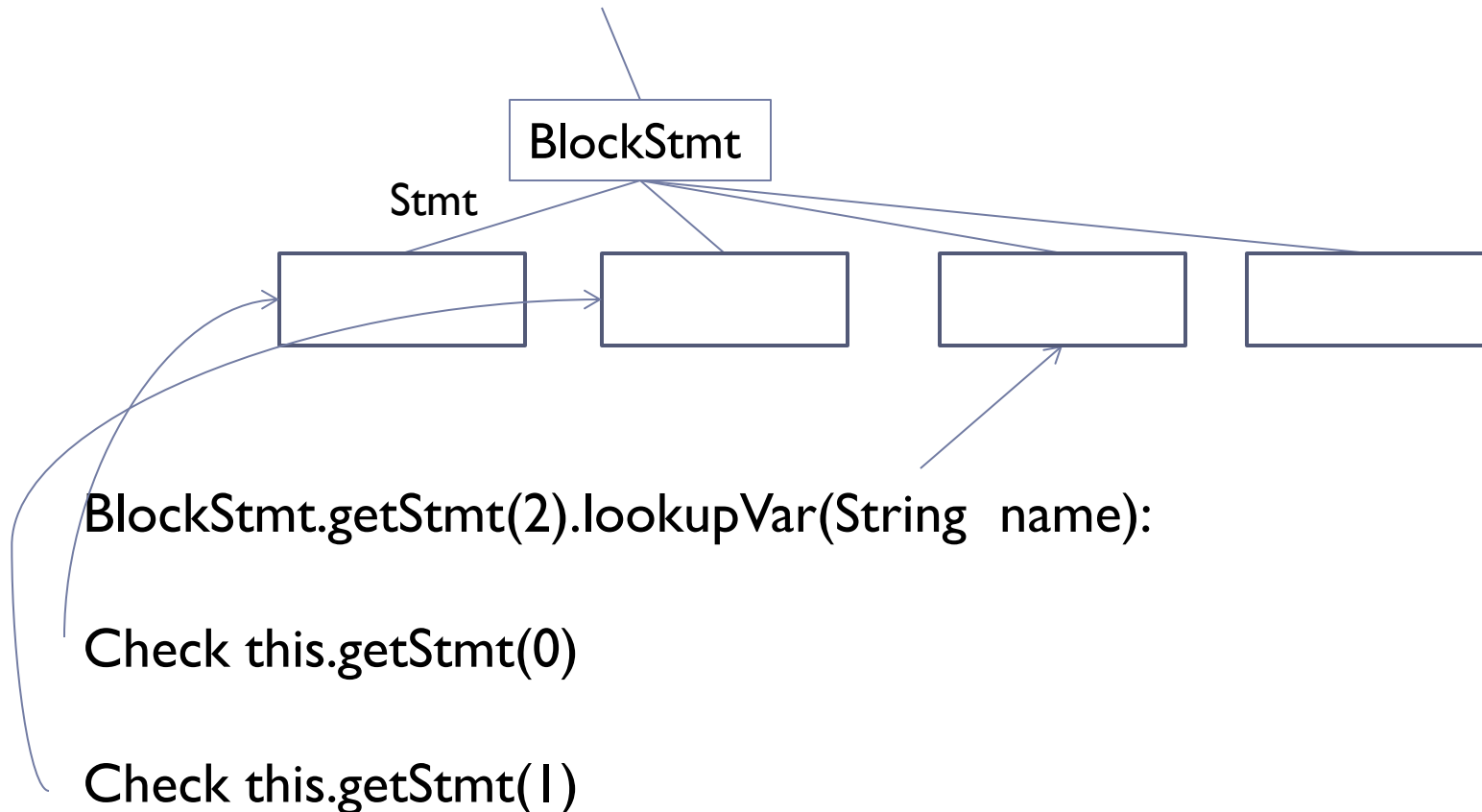
**Note that
VarDecl has to
be found in a
Stmt before
Stmt i.**

**Lookup in the
parent node**

Examples of
parameterized
attributes:
Slide 50

back

Example



If `varDecl` node not found, or name not found, call `this.lookupVar(name)`

Example of lookupVar actions

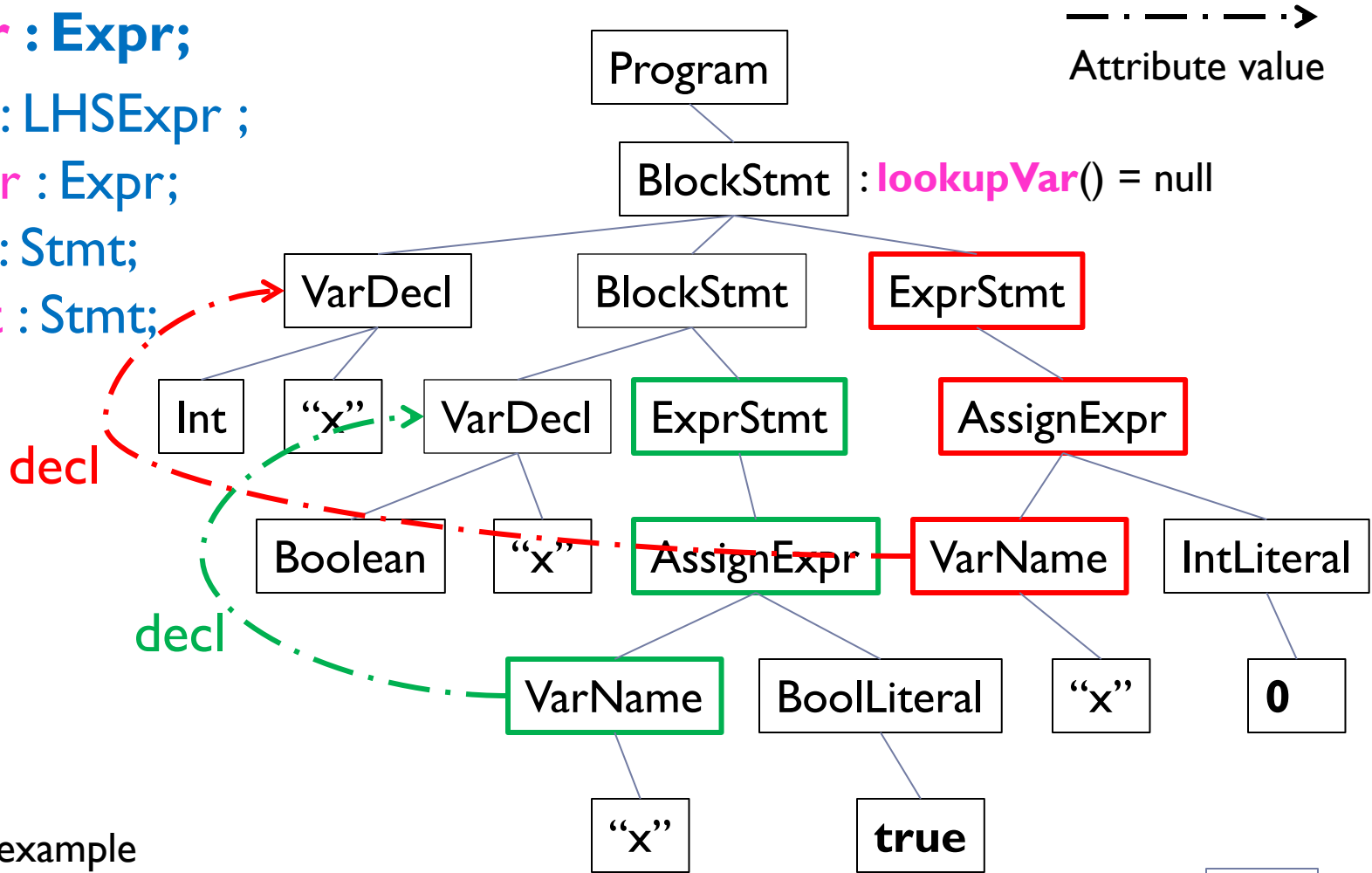
LHSExpr : Expr;

VarName : LHSExpr ;

AssignExpr : Expr;

ExprStmt : Stmt;

BlockStmt : Stmt;



Another example

back

Basic type analysis

- ▶ This is the problem of determining the type of an expression in the language.
- ▶ We define a synthesised attribute **type** on the node type *Expr* that compute the type of an expression node. Why?
- ▶ The type of an expression is described by a **type descriptor**. Type descriptors can be defined by **abstract grammar** in the **.ast** file. E.g.

```
abstract TypeDescriptor;
```

```
IntType : TypeDescriptor;
```

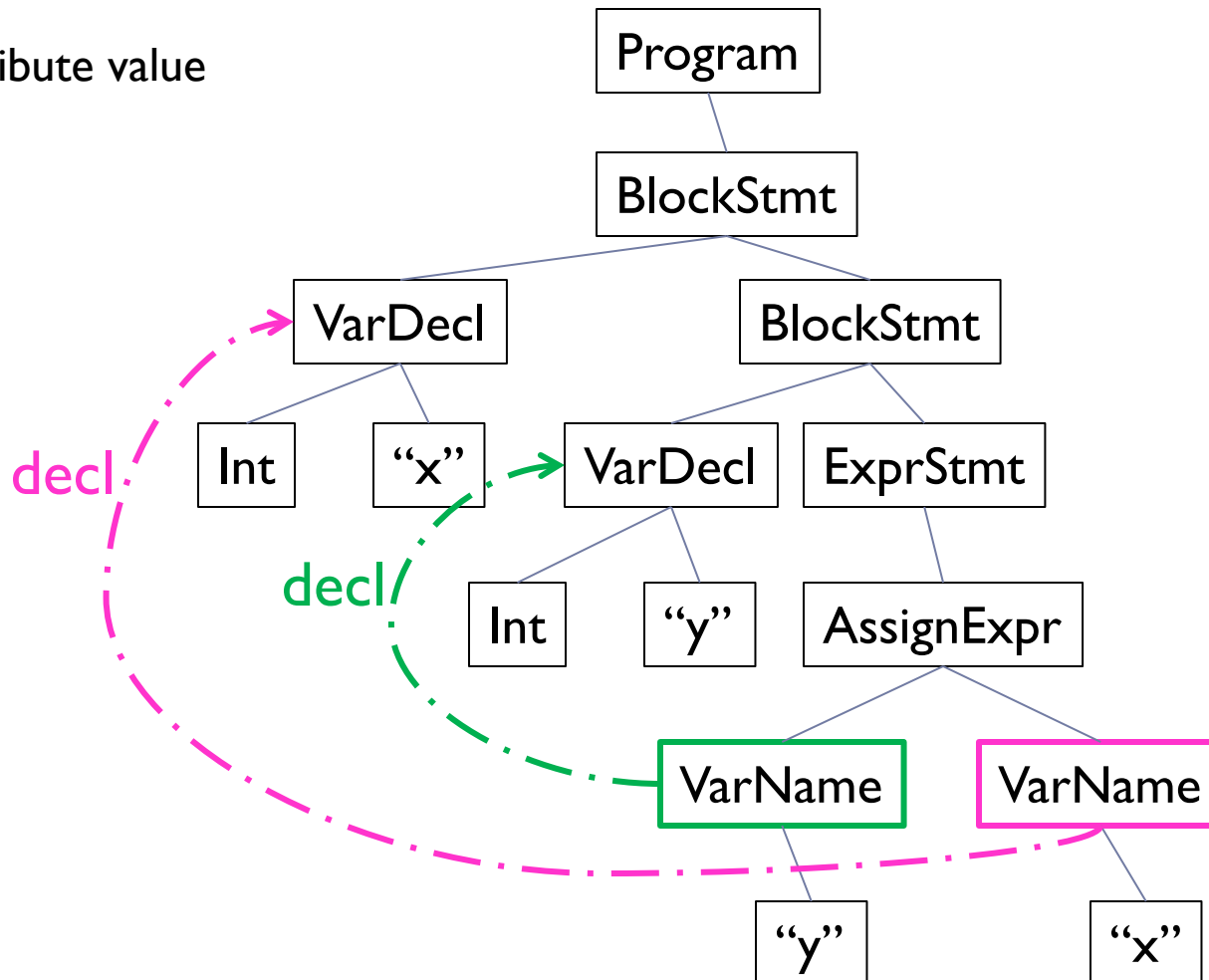
```
BooleanType : TypeDescriptor;
```

```
ArrayType : TypeDescriptor ::= ElementType:TypeDescriptor;
```

Slide 54

back

Why the node type typeDescriptor is needed



```
{ int x;
  { int y;
    y = x;
  }
}
```

The type descriptor of a type name

- ▶ Each type name has a type descriptor.
- ▶ For “Int” and “Boolean”, they just need a single instance of IntType and BooleanType.
- ▶ The type descriptor provides two fields to reference the single instances of IntType and BooleanType. These are inter-type field declarations made in an aspect in **.jrag** file.

Slide 54

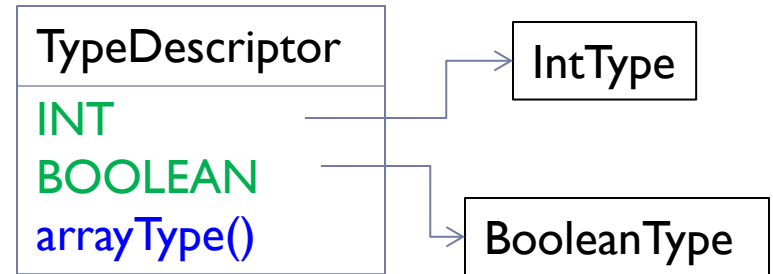
```
public static IntType TypeDescriptor.INT = new IntType();  
public static BooleanType TypeDescriptor.BOOLEAN = new  
    BooleanType();  
syn lazy ArrayType TypeDescriptor.arrayType() = new  
    ArrayType(this);
```

Slide 64

back

Java classes for the type descriptors

```
public abstract class TypeDescriptor {  
    public static IntType INT = new IntType();  
    public static BooleanType BOOLEAN = new BooleanType();  
    public ArrayType arrayType() {  
        return new ArrayType(this);  
    }  
    ....  
}  
  
public class IntType extends TypeDescriptor { ....}  
public class BooleanType extends TypeDescriptor { ....}  
public class ArrayType extends TypeDescriptor {  
    private final TypeDescriptor ElementType;  
    public ArrayType(TypeDescriptor ElementType) {  
        this.ElementType = ElementType;  
    }  
    ....  
}
```



Slide 64

back

Back to Slide 69

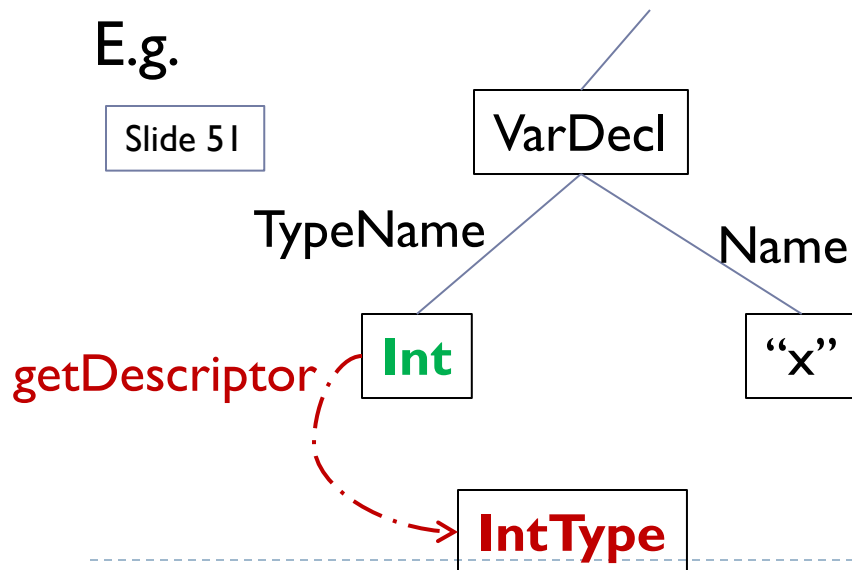
The type descriptor of a type name Int

```
syn lazy TypeDescriptor TypeName.getDescriptor();  
eq Int.getDescriptor() = TypeDescriptor.INT;  
eq Boolean.getDescriptor() = TypeDescriptor.BOOLEAN;  
eq ArrayTypeName.getDescriptor() =  
    getElementType().getDescriptor().arrayType();
```

Slide 54

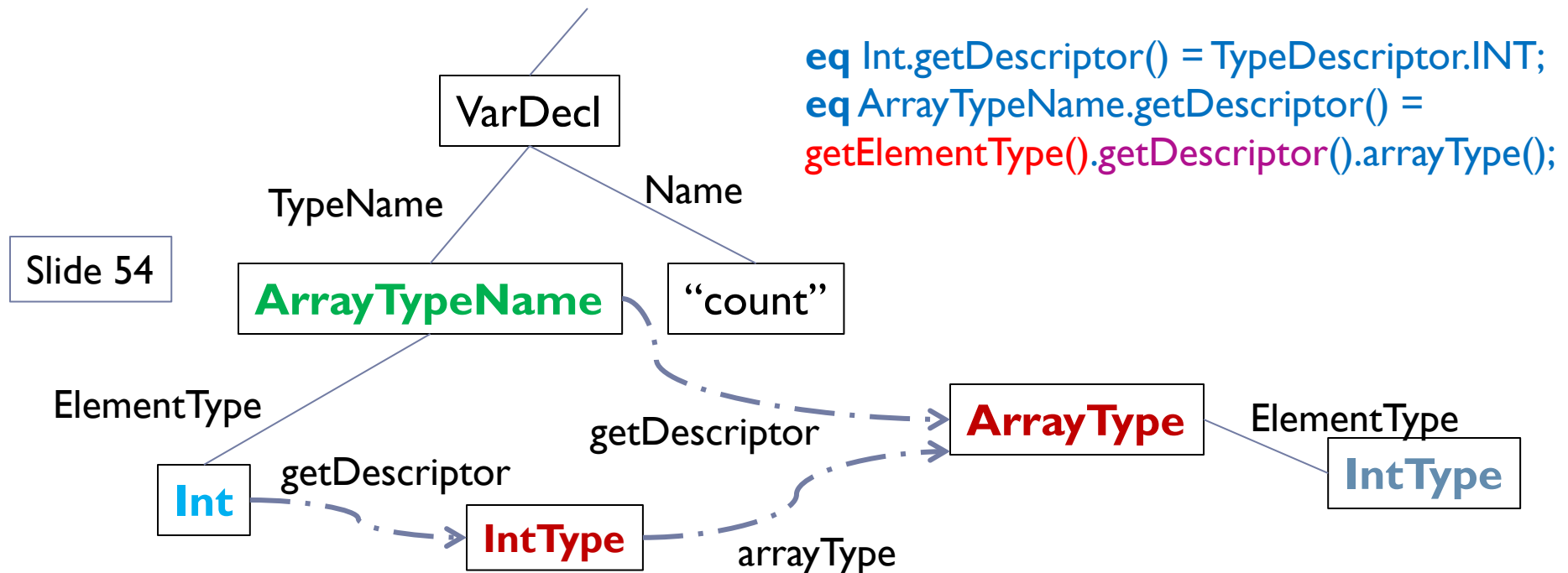
E.g.

Slide 51



Int.getDescriptor()
will return the type
descriptor **IntType**

The type descriptor of a type name ArrayTypeName



Firstly, `getDescriptor()` for node **ArrayTypeName** will call `getElementType()` which returns **Int**

Secondly, `Int.getDescriptor()` returns a type descriptor **IntType**.

Finally, `IntType.arrayType()` returns **ArrayType** with child **IntType**.

The type of an expression

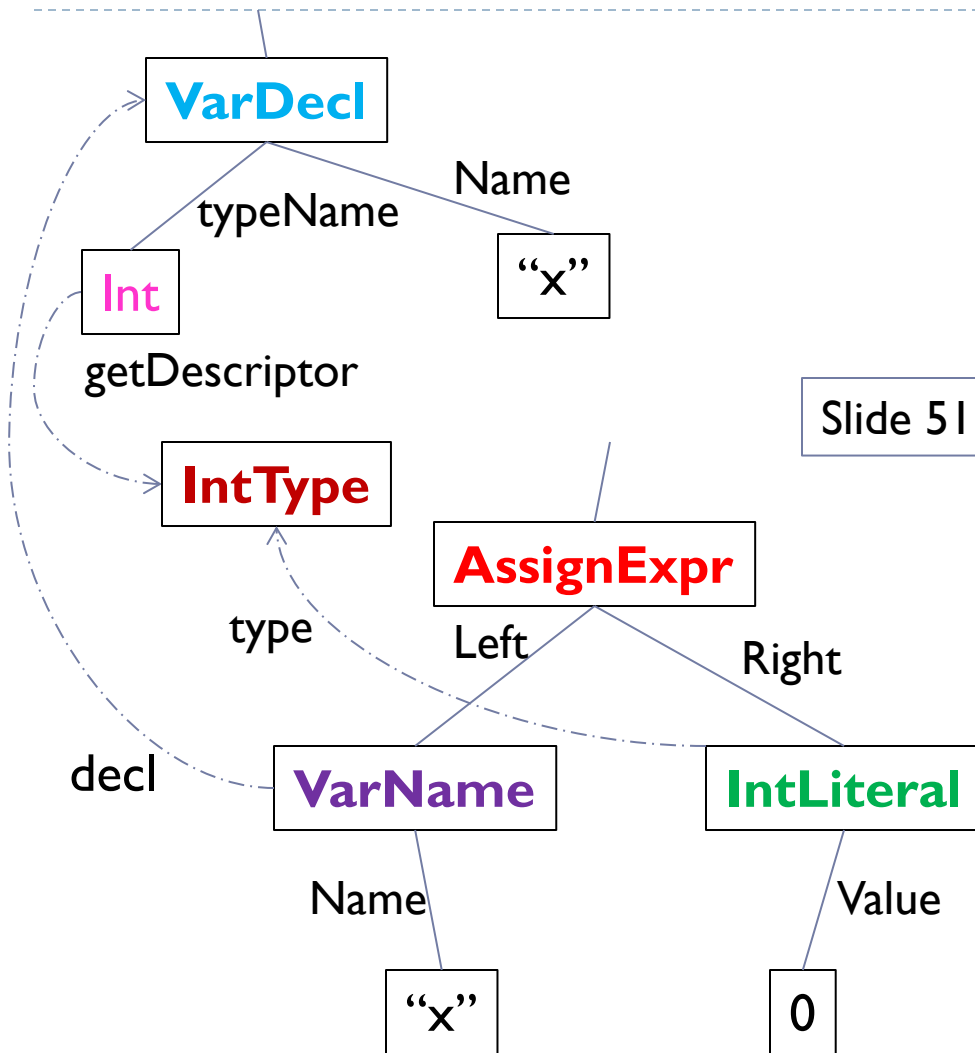
- ▶ Every expression node needs to compute its type.

Slide 53

```
syn TypeDescriptor Expr.type();  
eq IntLiteral.type() = TypeDescriptor.INT;  
eq BoolLiteral.type() = TypeDescriptor.BOOLEAN;  
eq ArrayLiteral.type() = getExpr(0).type().arrayType();  
eq VarName.type() =  
    decl().getTypeName().getDescriptor();  
eq ArrayIndex.type() =  
    ((ArrayType)getBase().type()).getElementType();  
eq AssignExpr.type() = getRight().type();  
eq BinaryExpr.type() = TypeDescriptor.INT;
```

back

The type of an expression



IntLiteral.type() returns the type descriptor **IntType**.

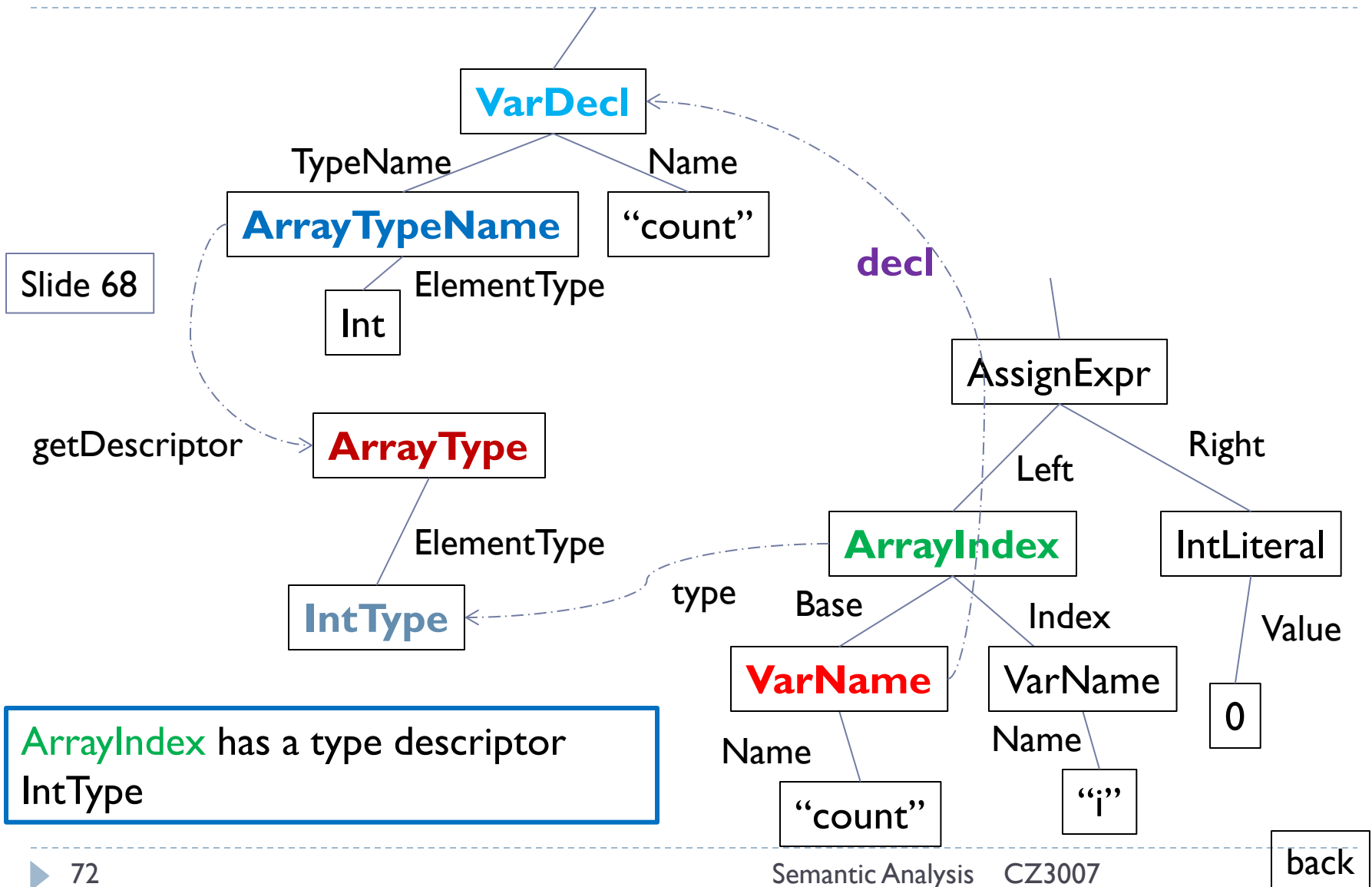
AssignExpr.type() returns the type descriptor **IntType**.

VarName.type() : (= **decl()**.
getTypeName().getDescriptor
();)

1. **decl()** returns the reference to **VarDecl**;
2. **VarDecl.getTypeName()** returns TypeName **Int**;
3. **Int.getDescriptor()** returns the type descriptor **IntType**

The type of an expression (count[i])

```
ArrayIndex.type()=((ArrayType)getBase().type()).getElementType();
```



Evaluation steps

1. **ArrayIndex.type()** : **getBase()** returns the reference to node **VarName**;
2. **VarName.type()** : **decl()** returns the reference to **VarDecl**;
3. **VarDecl.getTypeName()** returns the reference to **ArrayTypeName**;
4. **ArrayTypeName.getDescriptor()** returns the type descriptor **ArrayType** (as in Slide 67).
5. **ArrayType.getElementType()** returns **IntType**.

eq **ArrayIndex.type()** =

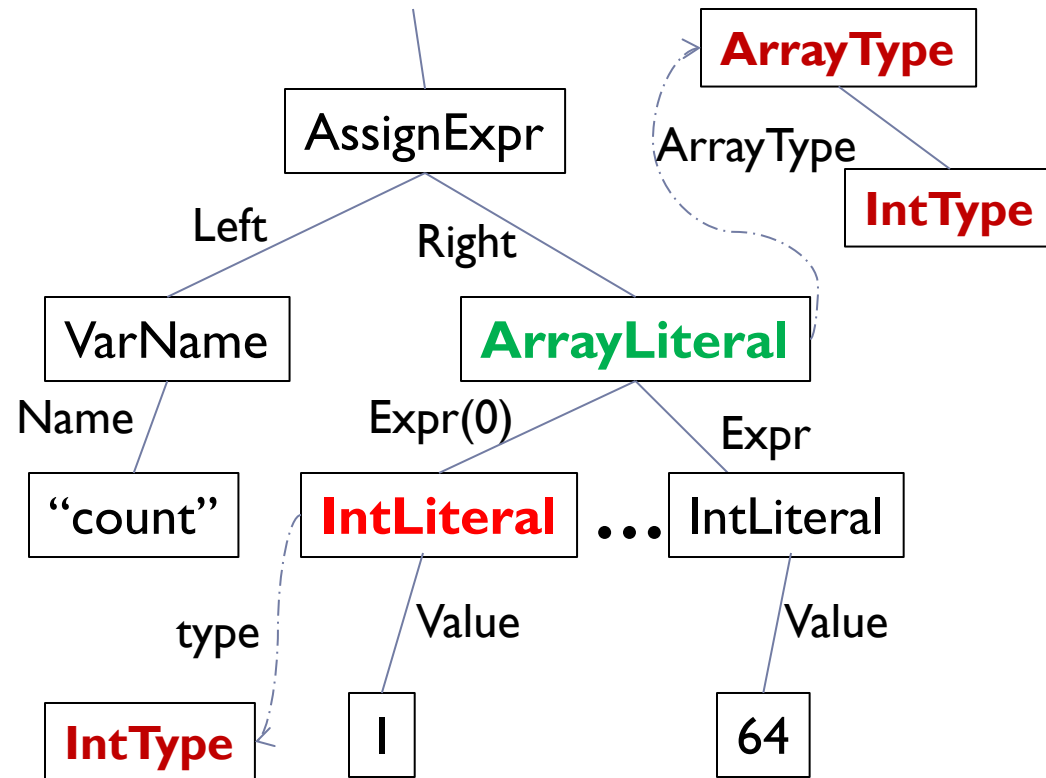
((ArrayType)getBase().type()).getElementType();

eq **VarName.type()** = **decl().getTypeName().getDescriptor();**

The type of an expression ($\{1, \dots, 64\}$)

ArrayLiteral.type() :

1. `getExpr(0)` returns the reference to the first **IntLiteral** node;
2. **IntLiteral.type()** returns the type descriptor **IntType**; Slide 69
3. **IntType.arrayType()** returns a type descriptor **ArrayType** with child **IntType** (illustrated in slide 68).



back

eq `ArrayLiteral.type() = getExpr(0).type().arrayType();`

- ▶ Note that none of these equations performs any error checking. E.g.

- ▶ `BinExpr.type()` simply returns `TypeDescriptor.INT` without checking the operands.

Slide 70

- ▶ `AssignExpr.type()` does not check whether the LHS and RHS are type compatible.

- ▶ These checks are done by separate semantic error checking code to be presented next.
- ▶ However, checking of some other type errors are needed in type analysis of a working compiler. E.g.
 - ▶ `ArrayIndex.type()` does not check whether we have an `ArrayType`.
 - ▶ `VarName.type()` does not check whether `decl()` returns `null`.

More advanced name and type analysis

- ▶ To scale our simple name analysis to a full real-world programming language like Java, several extensions are needed.
- ▶ Many languages have separate namespaces for variables, types and functions/methods. Hence, we need to define separate lookup attributes for these namespaces, such as `lookupMethod`.

Slide 61
- ▶ Another feature of object-oriented languages that name analysis needs to handle is inheritance: a class have its own fields, together with fields inherited from its super class.

Slide 16
- ▶ A number of attributes may be defined in addition to what we have seen so far.

More advanced name and type analysis

- ▶ A synthesised attribute *lookupMemberField* may be defined to first invoke attribute *lookupLocalField* to check whether the field is defined locally. Slide 61
- ▶ If it is not, it uses attribute *lookupType* to find the declaration of the super class, and recursively invokes *lookupMemberField* on the super class to see if the field is defined there.
- ▶ We also need to handle user-defined types, i.e., classes and interfaces. These types are defined by type declaration. Slide 54
- ▶ To determine what type declaration such a type name refers to, we need to implement an attribute *lookupType* to look up a type name from a given point in the AST. Also, we need to define a new kind of type descriptor that wraps a class or interface declaration.

Semantic Error Checking

- ▶ A program may be syntactically well-formed, but still exhibit semantic errors.
- ▶ What semantic errors the compiler should check for depends on the language.
- ▶ Common semantic errors checked by a compiler include variables used in an expression should be in scope, no two fields in the same class should have the same name, etc.
- ▶ Semantic error checking is implemented by defining a method **check** for every statement and expression type.

Scope checking for our simple language

- ▶ Scope checking is performed by the check methods on node types **VarDecl** and **VarName**.
- ▶ For **VarName**, we simply need to ensure that a declaration of that name is in scope.
- ▶ For variable declarations, we want to check that there isn't another variable of the same name declared within the same block.

Scope checking for our simple language (inter-type methods in .jrag file)

```
public void VarDecl.namecheck() {
```

Slide 59

```
    VarDecl conflicting = lookupVar(getName());
```

```
    if (conflicting != null && conflicting.getParent() ==  
                                             this.getParent())
```

Slide 61

```
        error("duplicate local variable with name" +  
              getName());
```

```
}
```

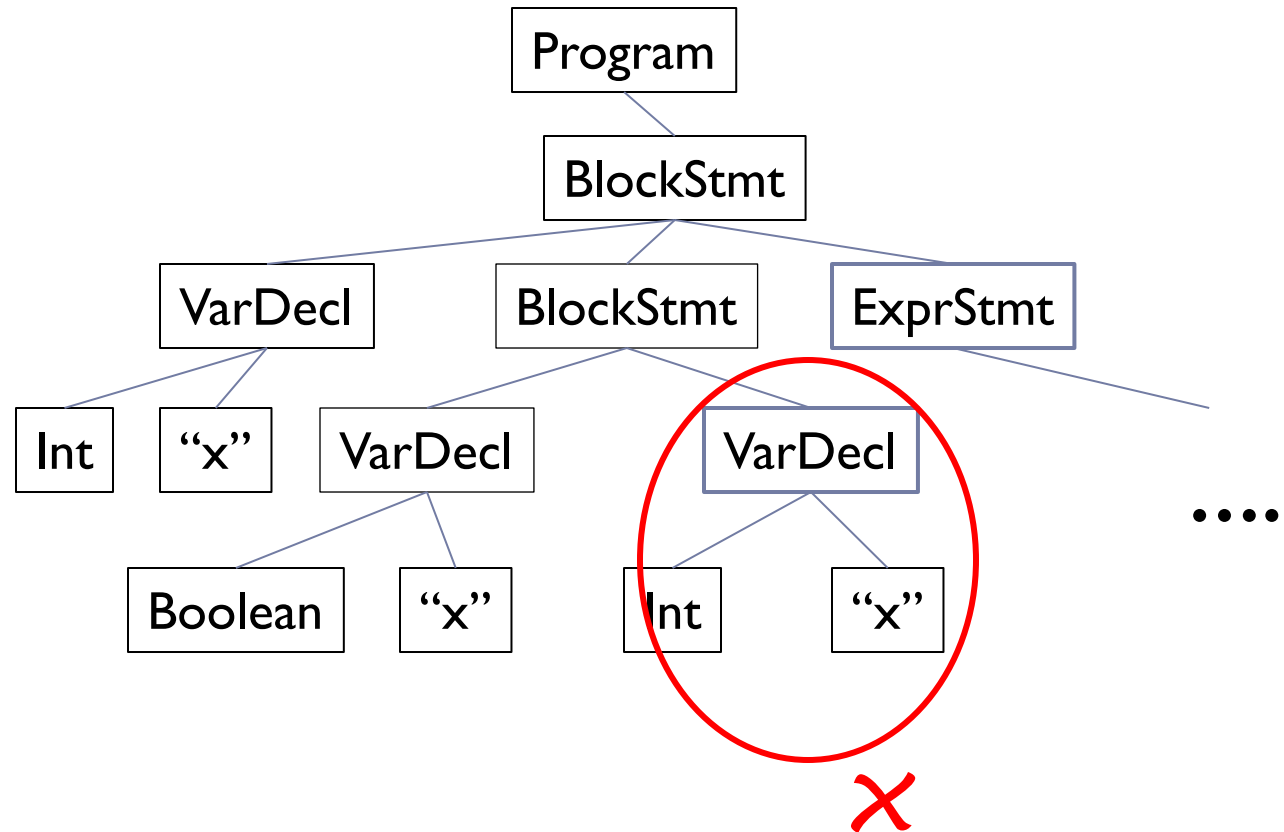
```
public void VarName.namecheck() {
```

```
    if (decl() == null)
```

```
        error("undeclared variable " + getName());
```

```
}
```


Example of **VarDecl.namecheck()**



Type checking for our simple language (inter-type methods in .jrag file)

```
public void ArrayLiteral.typecheck() {  
    // recursively check every expression  
    for (Expr expr : getExprs())    expr.typecheck();  
    if (getNumExpr() == 0) { // forbid empty array literals  
        error("empty array literals not allowed");  
    } else { // all expressions should have the same type  
        TypeDescriptor tp = getExpr(0).type();  
        for (int i = 1; i < getNumExpr(); ++i)  
            if (getExpr(i).type() != tp)  
                error("inconsistent types");  
    }  
}
```

Slide 73

Type checking for our simple language
(inter-type methods in .jrag file)

```
public void ArrayIndex.typecheck() {  
    // first recursively check the base and the index  
    getBase().typecheck();  
    getIndex().typecheck();  
    // ensure that the base expression's type is an array type  
    if (!(getBase().type() instanceof ArrayTypeDescriptor))  
        error("array index base must be an array");  
    // ensure that the index expression has type int  
    if (getIndex().type() != TypeDescriptor.INT)  
        error("array index must be numeric");  
}
```

Slide 71

Type checking for our simple language (inter-type methods in .jrag file)

```
public void AssignExpr.typecheck() {  
    // first recursively check the LHS and RHS  
    getLeft().typecheck();  
    getRight().typecheck();  
    // LHS and RHS have the same type  
    if (getRight().type() != getLeft().type())  
        error("LHS and RHS types do not match");  
}
```

Type checking for our simple language (inter-type methods in .jrag file)

```
public void BinaryExpr.typecheck() {  
    // first recursively check the left and right expression  
    getLeft().typecheck();  
    getRight().typecheck();  
    // Both the left and the right should be Int type  
    if (getLeft().type() != TypeDescriptor.INT ||  
        getRight().type() != TypeDescriptor.INT)  
        error("both operands must be integers");  
}
```

Type checking for our simple language (inter-type methods in .jrag file)

- ▶ In the error checking for statements, we recursively check their child statements and expressions.
- ▶ We ensure that the test expression in an **if** statement and the loop condition in a **while** loop have type boolean.

```
public void Program.typecheck() {  
    getBlockStmt().typecheck();  
}
```

Slide 52

```
public void BlockStmt.typecheck() {  
    for (Stmt stmt : getStmts())    stmt.typecheck();  
}
```

Type checking for our simple language (inter-type methods in .jrag file)

```
public void IfStmt.typecheck() {  
    getExpr().typecheck();  
    getThen().typecheck();  
    if (hasElse())    getElse().typecheck();  
    if (getExpr().type() != TypeDescriptor.BOOLEAN)  
        error("if condition must be boolean");  
}
```

Type checking for our simple language (inter-type methods in .jrag file)

```
public void WhileStmt.typecheck() {  
    getExpr().typecheck();  
    getBody().typecheck();  
    if (getExpr().type() != TypeDescriptor.BOOLEAN)  
        error("loop condition must be boolean");  
}
```

```
public void ExprStmt.typecheck() {  
    getExpr().typecheck();  
}
```


Finally

- ▶ **.ast** file contains the abstract grammar to define all the node types in the AST, including the type descriptors.
- ▶ **.jrag** files contain
 - ▶ the attribute grammar to define all the attributes.
 - ▶ all the inter-type declarations, including the inter-type field declarations (e.g. slide 65) and method declarations (e.g. check()).
- ▶ It is better to group inter-type declarations and attribute grammars that logically belong together in one aspect in one .jrag file



Some review questions/tasks

1. What does a semantic analyzer do? What is its input?
2. What is an Abstract Grammar used for?
3. What is an Abstract Syntax Tree for a program?
4. Which part of a compiler generates/builds the AST for a program?
5. Follow the semantic actions in the Beaver specification on slides 25 to show how the ASTs for the Expr “ $2*(3+4)$ ” and “ $2*3+4$ ” are built respectively.
6. What is an Attribute Grammar used for in building a compiler?
7. For the AST on slide 33, what checks are done with respect to name checking and type checking (rf: slide 28)?