Compiler Techniques

4. Semantic Analysis

Huang Shell Ying

The Compiler So Far

- Lexical analysis: Detects illegal characters/strings
- Parsing: Detects syntax errors
- Semantic analysis: Catches all remaining errors
  - Last "front end" phase

Ref: 1_Introduction
Structure of a Compiler

- Why a separate semantic analysis?
  - The semantic rules cannot be expressed with context-free grammars
  - Separation of concerns
  - To make the compiler easier to understand, debug, improve, maintain and extend

What Does Semantic Analysis Do?

- Checks whether semantic rules are violated:
  - Scoping errors
    - A variable name is not declared in its scope.
    - Methods/functions called are not declared.
    - Variable/method/class names are declared more than once in a scope.
  - Type inconsistencies
    - Operators and operands are not type-compatible.
    - Functions/methods are called with incorrect number of parameters and/or types.
    - Return expression has a type not compatible with function type.
  - And others . . .
- Semantic rules depend on the language.

Slide 119 provides links to scope rules of some languages

Why are these checks necessary?

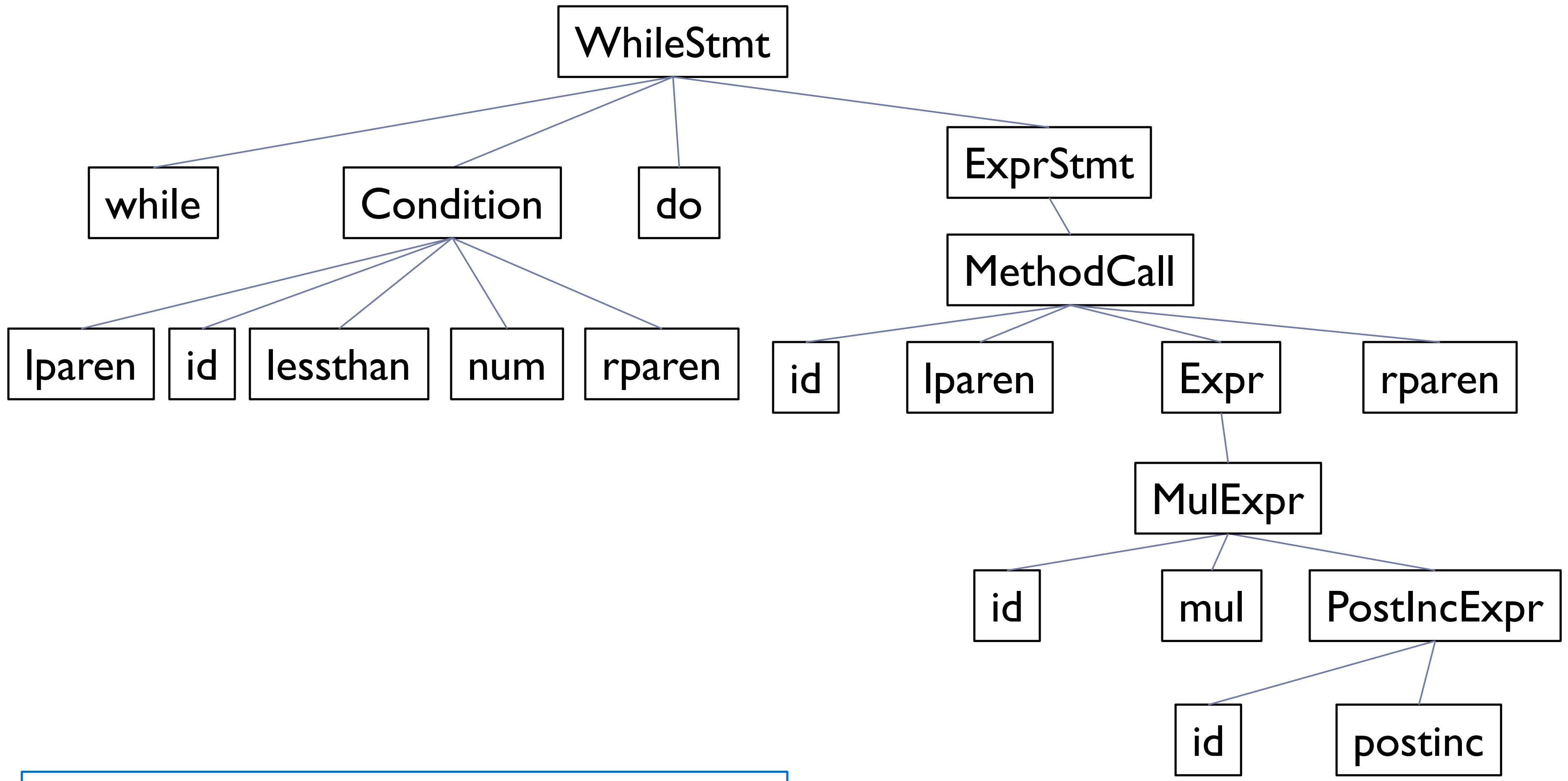
Abstract Syntax Tree (AST)

- The AST is a central data structure for all post-parsing activities like semantic checking, code generation, etc.
- Parse trees show the structure of sentences but they often have redundant information.
- The AST catches the source program structure, omitting the unnecessary syntactic details.
- The **parser** recognises 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

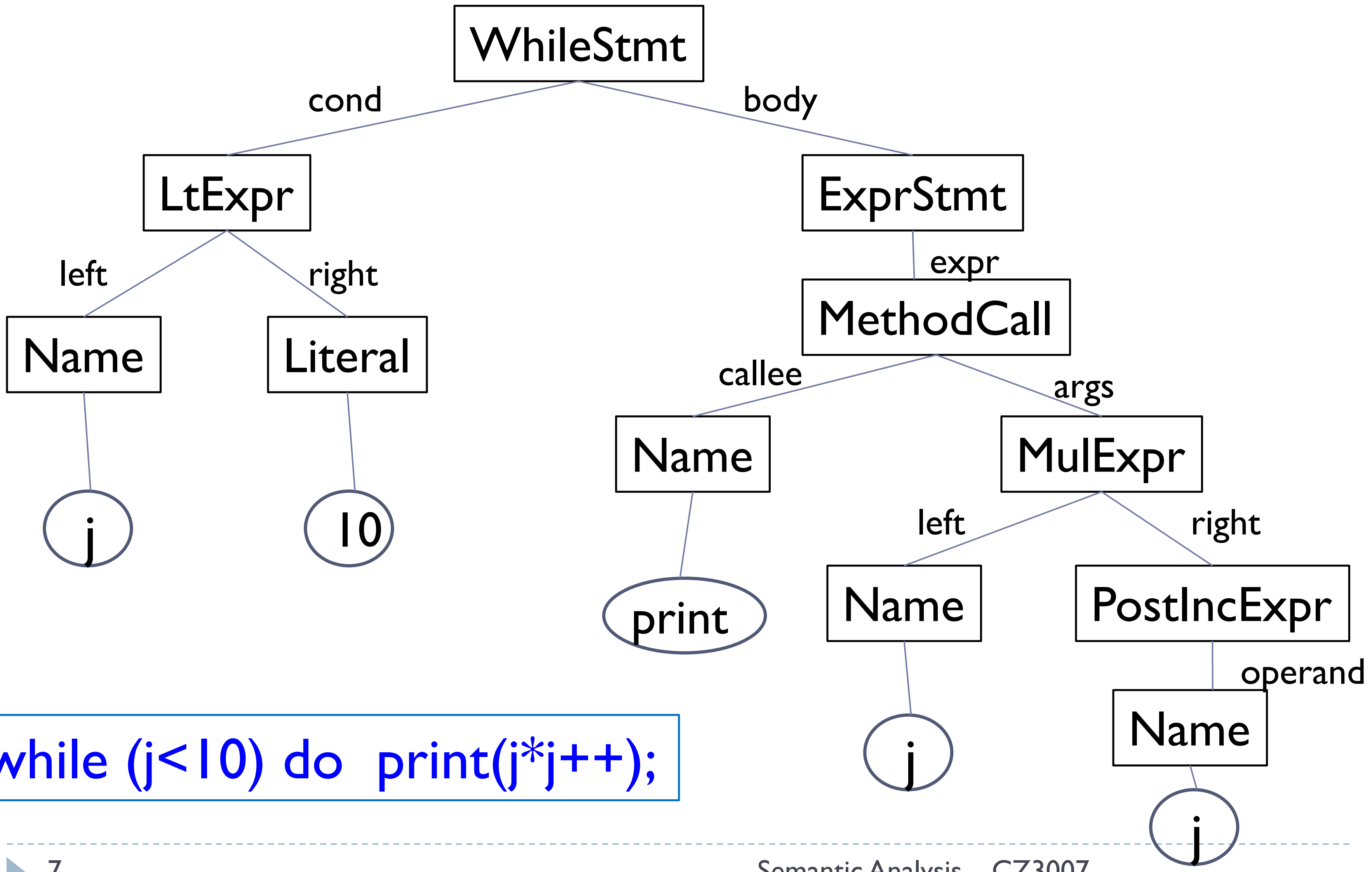
- The role and purpose of the AST and parse trees are different:
  - A parse tree represents a derivation—a conceptual tool for analysing the parsing process. E.g., next slide
  - An AST is a real data structure and stores information for semantic analysis and code generation. E.g., slide 7
- The tree 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



```
while (j<10) do  print(j*j++);
```

Example of an AST



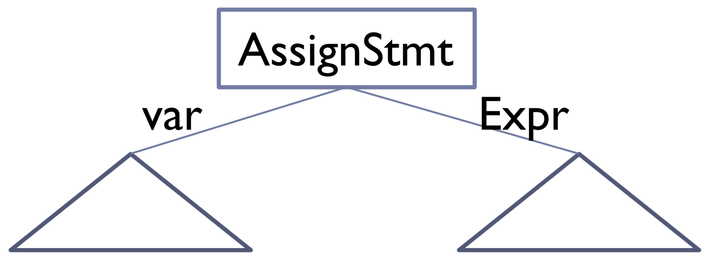
while (j<10) do print(j*j++);

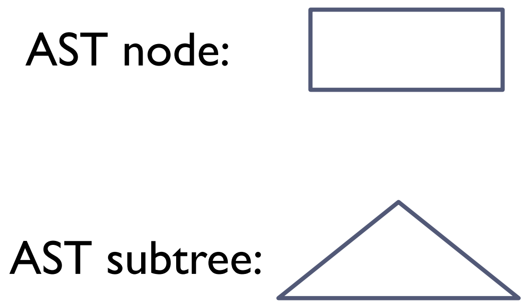
Design of the AST

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

Examples

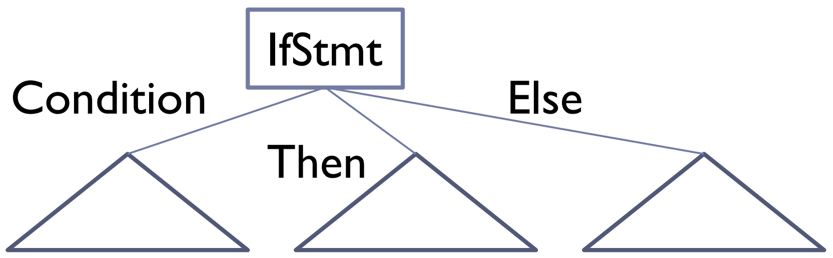
- Assignment statement: The target of the assignment and the value to be stored at that target

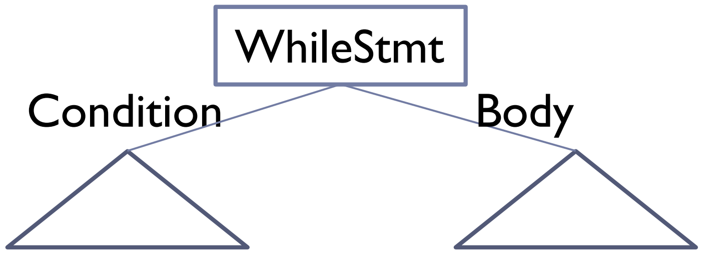




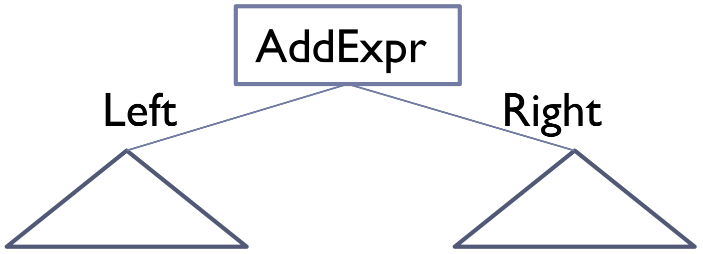
Design of the AST

- 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



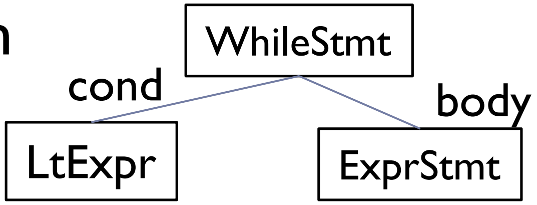


- Arithmetic expression: Add

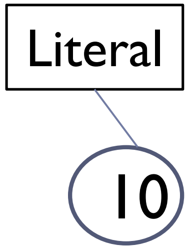


AST

- Every node in the AST is labelled by a *node type*. E.g. $\boxed{\text{WhileStmt}}$
- 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. E.g.



- 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. E.g.



- When a compiler is implemented in an object-oriented programming language, each node type is represented by a class. Each edge from a parent to a child node is represented by a field variable in the parent class.

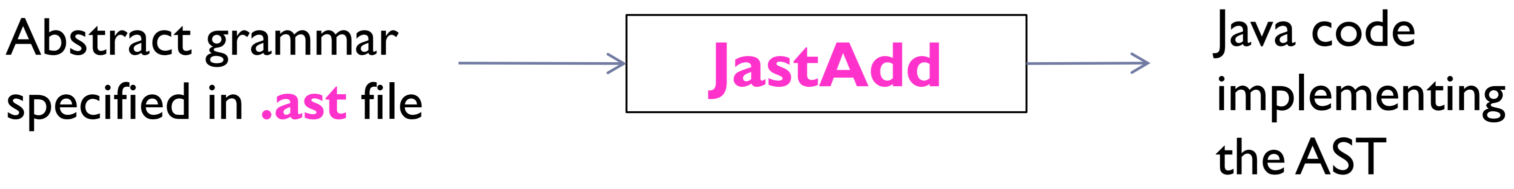
AST

- These classes are called *AST classes*.
- A number of AST classes may have common structure and behaviours so **a class hierarchy will be created** where one class can inherit part or all of its structure and behaviour from a super/abstract class.

  E.g., Both AddExpr and SubExpr have left and right operands.
- There may be more than one level of super/abstract classes in the class hierarchy depending on the sharing of common behaviours among classes.
- The structure of ASTs can be described by an ***abstract grammar***.

Abstract Grammars

- The translation from abstract grammars to Java classes is implemented by the **JastAdd** tool (JastAdd: *just add* to the *ast*) .


Abstract grammar specified in **.ast** file
JastAdd
Java code implementing the AST

- Abstract grammars can easily be translated into sets of Java classes, with every node type represented by one Java class.
- Abstract grammars define **sets of trees** (namely, abstract syntax trees), while context-free grammars define **sets of sentences**.

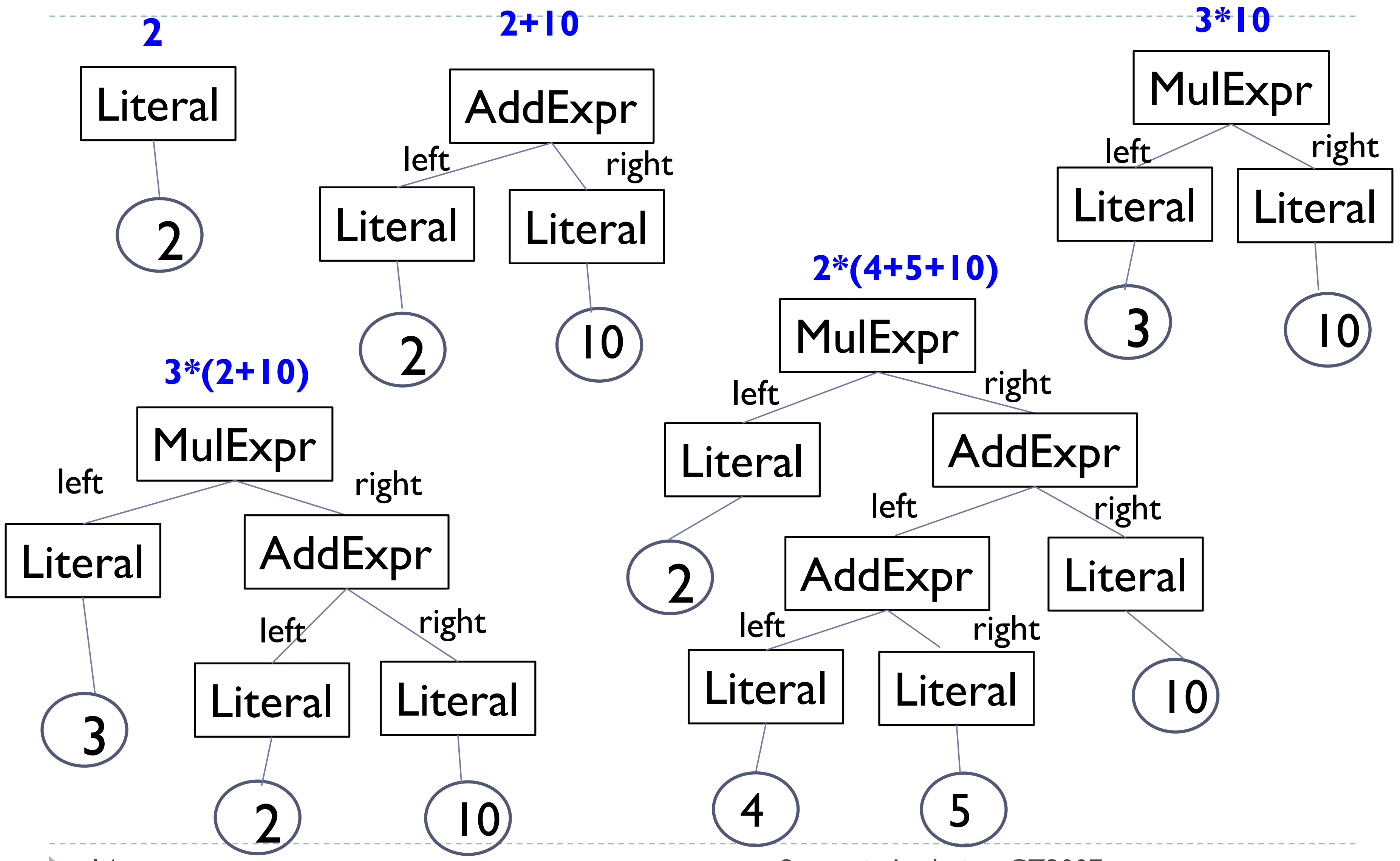
Abstract Grammars

- 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 an arithmetic expression with addition and multiplication:

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

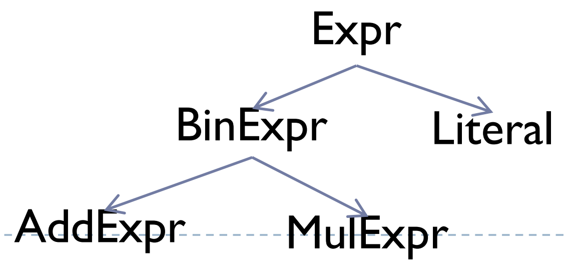
Examples of ASTs of Expressions



Observations

1) Three node types appear in the ASTs: AddExpr, MulExpr and Literal.
2) AddExpr and MulExpr are nonterminal nodes while Literal is a terminal node.
3) AddExpr and MulExpr have a common structure; Reason for a superclass for them.
4) AddExpr, MulExpr and Literal have a common behavior — they all are specific types of expressions; Reason for a superclass for them

An Abstract Grammar



The **abstract grammar**:

```
1  abstract Expr;
2  abstract BinExpr : Expr ::= Left : Expr  Right : Expr ;
3  AddExpr : BinExpr;
4  MulExpr : BinExpr;
5  Literal : Expr ::= <Value : Integer> ;
```

Field of the class BinExpr

Type of the field

Class BinExpr inherits from Expr

The LHS and the RHS are separated by **::=**; LHS defines the class name RHS defines the fields.

ClassAddExpr inherits from BinExpr

- Abstract classes are useful to **structure inheritance hierarchies** and to **encapsulate common structure and behaviour** of their subclasses.
- Abstract classes do not represent real nodes in AST.

Java Classes Derived from the Abstract Grammar

```
// abstract Expr;
public abstract class Expr {}

// 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; }
}
```

Two fields of the class which are links to the children nodes.

Every class gets a constructor that creates a new AST node.

Every class has the get methods for accessing the fields.

Java Classes Derived from the Abstract Grammar

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

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

Java Classes Derived from the Abstract Grammar

```
// 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; }
}
```

A field of the class which is an integer variable.

Optional Children Nodes

- Often, some node types 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];
- 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:

```
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 *. E.g.,

```
BlockStmt : Stmt ::= Stmt*;
```

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

Java Class Derived from BlockStmt : Stmt ::= 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();
    }                                   }
```

Generating ASTs

- Once the Java classes for all AST node types have been generated using JastAdd, the parser can call the constructors of the AST classes to construct the AST.
- The semantic actions to create AST nodes will be placed in the grammar (CFG).
- The actions themselves are surrounded by **{:** and **:}** and have to be placed at the very end of a rule.
- Symbols appearing on the right hand side of a CFG rule can be named. The name in a nonterminal symbol is the link to the sub-AST built when the nonterminal is reduced. The name in a terminal symbol is the string value of the token. These names can be used in the semantic actions.

Semantic Actions Added to a Beaver Specification (Example)

CFG:

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

Abstract grammar:

```
abstract Expr;
abstract BinExpr : Expr ::= Left : Expr  Right : Expr ;
AddExpr : BinExpr;
MulExpr : BinExpr;
Literal : Expr ::= <Value : Integer> ;
```

Semantic Actions Added to a Beaver Specification (Example)

```
%terminals PLUS, MUL, NUMBER, LPAREN, RPAREN;
%goal Expr;
%typeof NUMBER = "String";
%typeof Expr, Term, Factor = "Expr";
Expr = Expr.e PLUS Term.t {:  return new AddExpr(e, t);  :}
        | Term.t {:  return t;  :}
        ;
Term = Term.t MUL Factor.f {:  return new MulExpr(t, f);  :}
        | Factor.f {:  return f;  :}
        ;
Factor = NUMBER.n {:  return new Literal(Integer.parseInt(n));  :}
        | LPAREN Expr.e RPAREN {:  return e;  :}
        ;
```

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

t is the AST obtained when the CFG rule for Term is reduced.

n is the string value of token NUMBER.

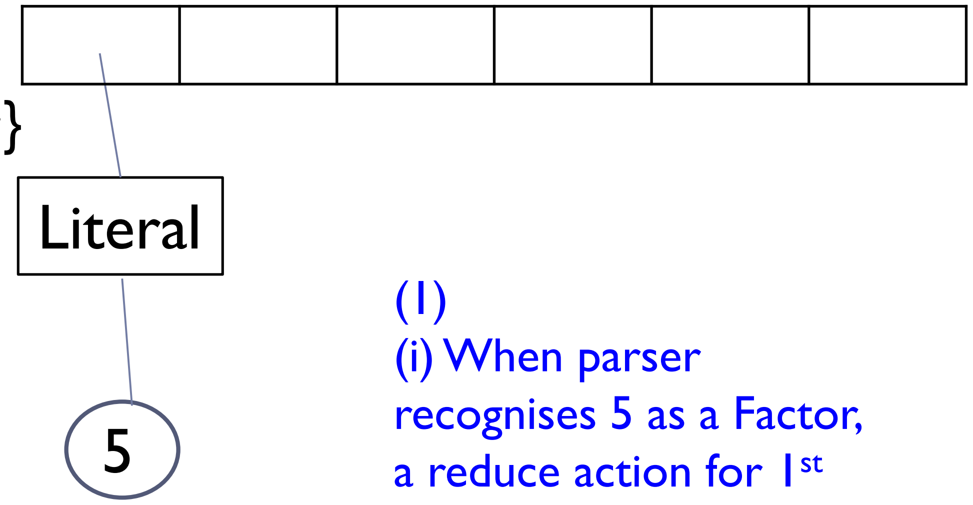
The Parser Builds the AST

- 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 (recap: LALR(1) parser, chapter 3), it will do pop the stack. Then execute the semantic action associated with that rule. An AST node is returned which is associated with the LHS of the rule being reduced.
- When the start nonterminal is reduced, the whole AST is built and returned.

Example of the AST for "5+4*3"

Stack without showing the states

```
Expr = Expr.e PLUS Term.t
    {: return new AddExpr(e, t); :}
        | Term.t
    {: return t; :}
        ;
Term = Term.t MUL Factor.f
    {: return new MulExpr(t, f); :}
        | Factor.f
    {: return f; :}
        ;
Factor = NUMBER.n
    {: return new Literal(Integer.parseInt(n)); :}
        | LPAREN Expr.e RPAREN
    {: return e; :}
        ;
```



(1)

(i) When parser recognises 5 as a Factor, a reduce action for 1st rule of Factor is done: Literal AST is created.

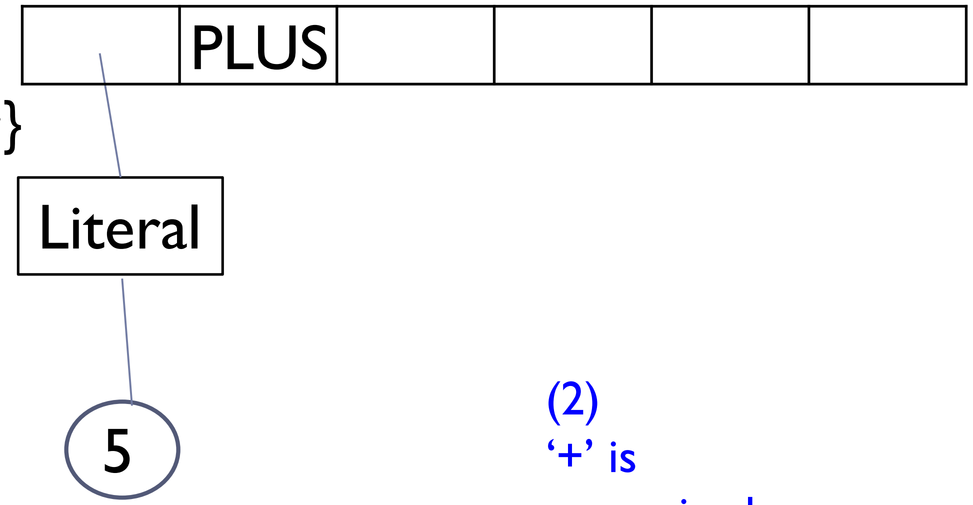
(ii) Factor is recognized as a Term which is recognized as an Expr.

(iii) The reference to the Literal AST is on the stack

Example of the AST for "5+4*3"

Stack without showing the states

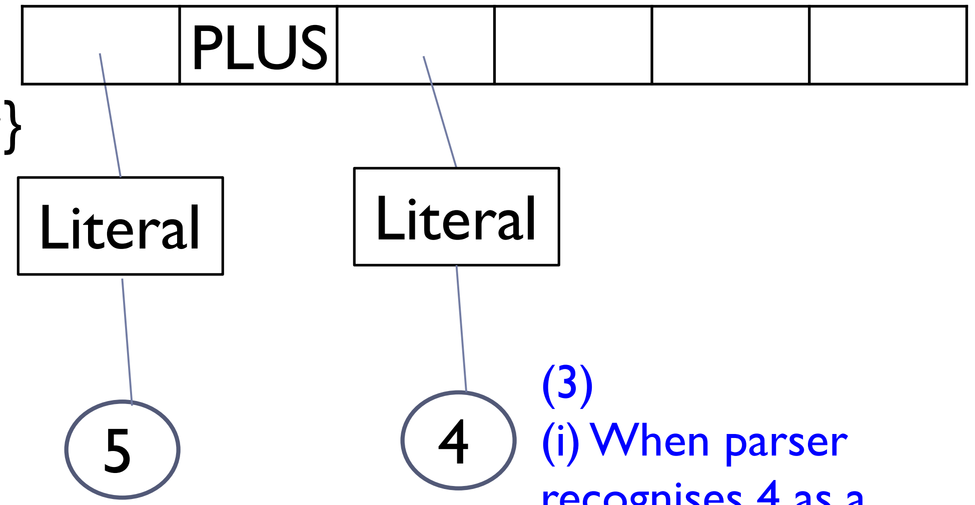
```
Expr = Expr.e PLUS Term.t
   {: return new AddExpr(e, t);  :}
      | Term.t
   {: return t;  :}
      ;
Term = Term.t MUL Factor.f
   {: return new MulExpr(t, f);  :}
      | Factor.f
   {: return f;  :}
      ;
Factor = NUMBER.n
   {: return new Literal(Integer.parseInt(n));  :}
      | LPAREN Expr.e RPAREN
   {: return e;  :}
      ;
```



(2)
'+' is recognised as PLUS which is pushed on the stack.

Example of the AST for “5+4*3”

Stack without showing the states



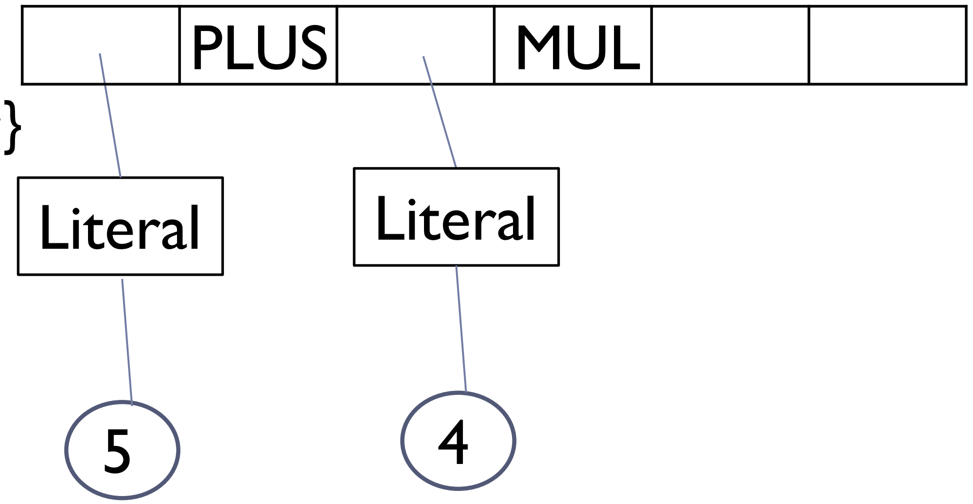
```
Expr = Expr.e PLUS Term.t
   {:  return new AddExpr(e, t);  :}
        | Term.t
   {:  return t;  :}
        ;
Term = Term.t MUL Factor.f
   {:  return new MulExpr(t, f);  :}
        | Factor.f
   {:  return f;  :}
        ;
Factor = NUMBER.n
   {:  return new Literal(Integer.parseInt(n));  :}
        | LPAREN Expr.e RPAREN
   {:  return e;  :}
        ;
```

(3)
(i) When parser recognises 4 as a Factor, a reduce action for 1st rule of Factor is done: Literal AST is created.
(ii) Factor is recognized as a Term.
(iii) The reference to the Literal AST is on the stack

Example of the AST for "5+4*3"

Stack without showing the states

```
Expr = Expr.e PLUS Term.t
    {: return new AddExpr(e, t); :}
        | Term.t
    {: return t; :}
        ;
Term = Term.t MUL Factor.f
    {: return new MulExpr(t, f); :}
        | Factor.f
    {: return f; :}
        ;
Factor = NUMBER.n
    {: return new Literal(Integer.parseInt(n)); :}
        | LPAREN Expr.e RPAREN
    {: return e; :}
        ;
```

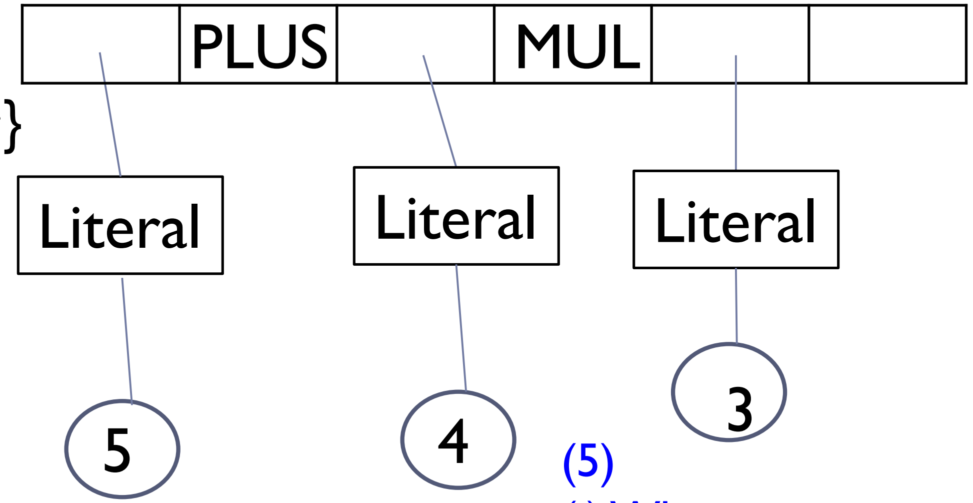


(4)
'*' is recognised as MUL which is pushed on stack.

Example of the AST for "5+4*3"

Stack without showing the states

```
Expr = Expr.e PLUS Term.t
    {: return new AddExpr(e, t); :}
        | Term.t
    {: return t; :}
        ;
Term = Term.t MUL Factor.f
    {: return new MulExpr(t, f); :}
        | Factor.f
    {: return f; :}
        ;
Factor = NUMBER.n
    {: return new Literal(Integer.parseInt(n)); :}
        | LPAREN Expr.e RPAREN
    {: return e; :}
        ;
```

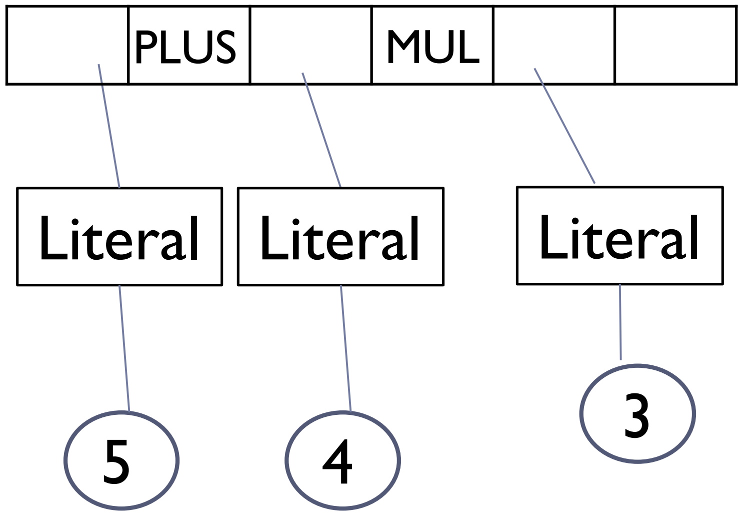


(5)
(i) When parser recognises 3 as a Factor, a reduce action for 1st rule of Factor is done. Literal AST is created.
(ii) The reference to the Literal AST is on the stack

Example of the AST for “5+4*3”

```
Expr = Expr.e PLUS Term.t
    {: return new AddExpr(e, t); :}
        | Term.t
    {: return t; :}
        ;
Term = Term.t MUL Factor.f
    {: return new MulExpr(t, f); :}
        | Factor.f
    {: return f; :}
        ;
Factor = NUMBER.n
    {: return new Literal(Integer.parseInt(n)); :}
        | LPAREN Expr.e RPAREN
    {: return e; :}
        ;
```

Stack without showing the states

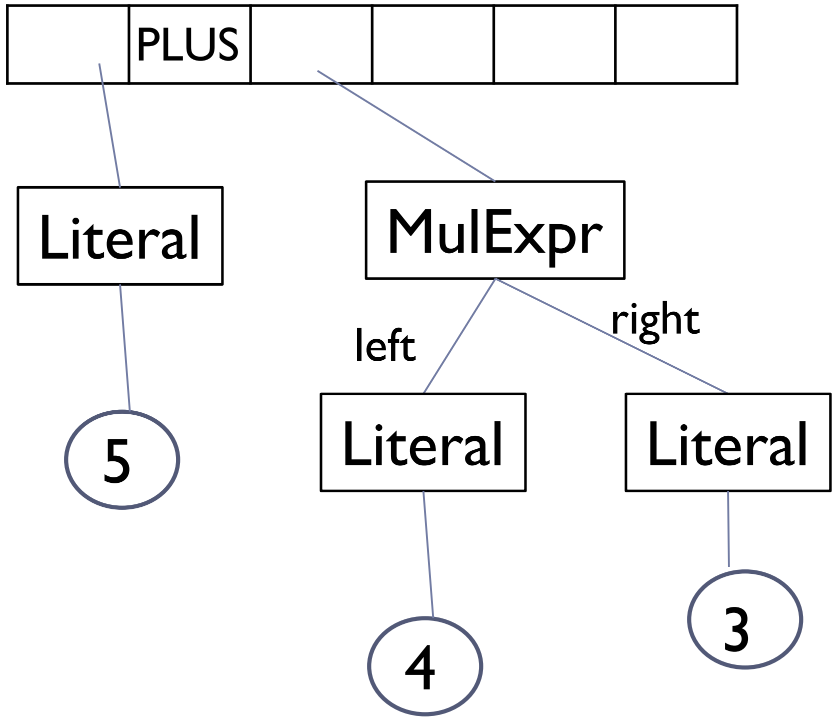


(6) (a)
When parser recognises Term * Factor as a Term, a reduce action for 1st rule of Term is done.

Example of the AST for “5+4*3”

Stack without showing the states

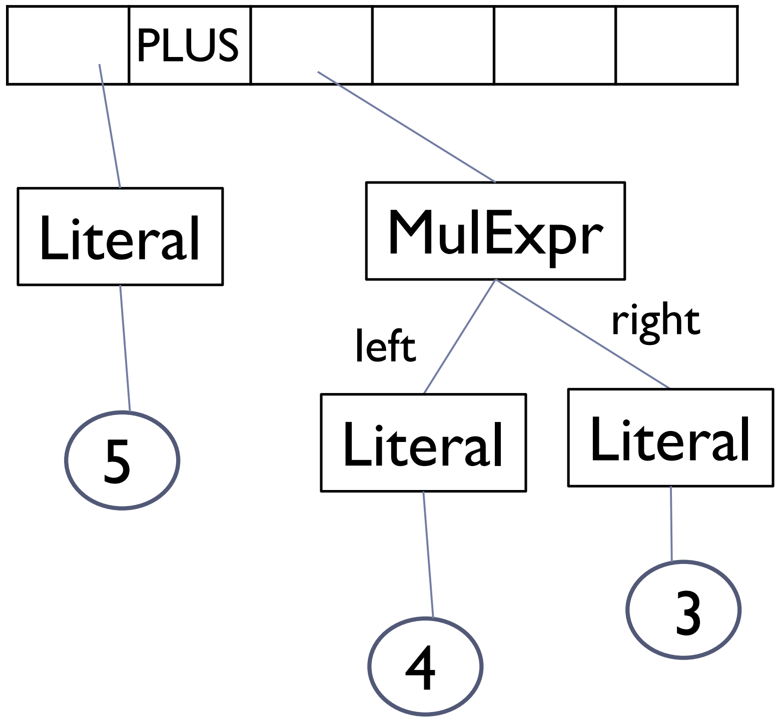
```
Expr = Expr.e PLUS Term.t
   {: return new AddExpr(e, t); :}
        | Term.t
   {: return t; :}
        ;
Term = Term.t MUL Factor.f
   {: return new MulExpr(t, f); :}
        | Factor.f
   {: return f; :}
        ;
Factor = NUMBER.n
   {: return new Literal(Integer.parseInt(n)); :}
        | LPAREN Expr.e RPAREN
   {: return e; :}
        ;
```



(6) (b)
Three items are popped off the stack. MulExpr AST is created and pushed on the stack.

Example of the AST for "5+4*3"

Stack without showing the states



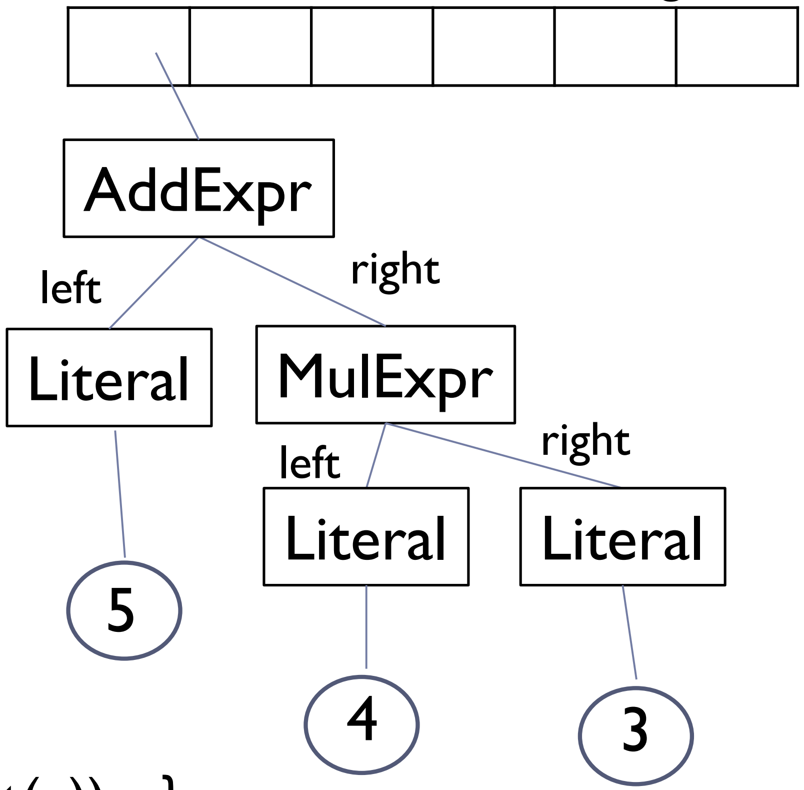
Expr = Expr.e PLUS Term.t
  {: **return new** AddExpr(e, t); :}
    | Term.t
  {: **return** t; :}
    ;
Term = Term.t MUL Factor.f
  {: **return new** MulExpr(t, f); :}
    | Factor.f
  {: **return** f; :}
    ;
Factor = NUMBER.n
  {: **return new** Literal(Integer.parseInt(n)); :}
    | LPAREN Expr.e RPAREN
  {: **return** e; :}
    ;

(7) (a)
When parser recognises Expr PLUS Term as an Expr, a reduce action for 1st rule of Expr is done.

Example of the AST for "5+4*3"

```
Expr = Expr.e PLUS Term.t
   {: return new AddExpr(e, t); :}
      | Term.t
   {: return t; :}
      ;
Term = Term.t MUL Factor.f
   {: return new MulExpr(t, f); :}
      | Factor.f
   {: return f; :}
      ;
Factor = NUMBER.n
   {: return new Literal(Integer.parseInt(n)); :}
      | LPAREN Expr.e RPAREN
   {: return e; :}
      ;
```

Stack without showing the states



(7) (b)
Three items are popped off the stack. AddExpr AST is created and pushed on the stack.

Test Yourself 4.1

1) The CFG defines a language of boolean expressions. Words in uppercase are tokens of the language.

```
BoolExpr = BoolExpr OR BoolTerm
         | BoolTerm
BoolTerm = BoolTerm AND BoolFactor
         | BoolFactor
BoolFactor = TRUE
         | FALSE
         | LPAREN BoolExpr RPAREN
         | NOT BoolFactor
```

Give the abstract grammar to define the AST nodes for the language.

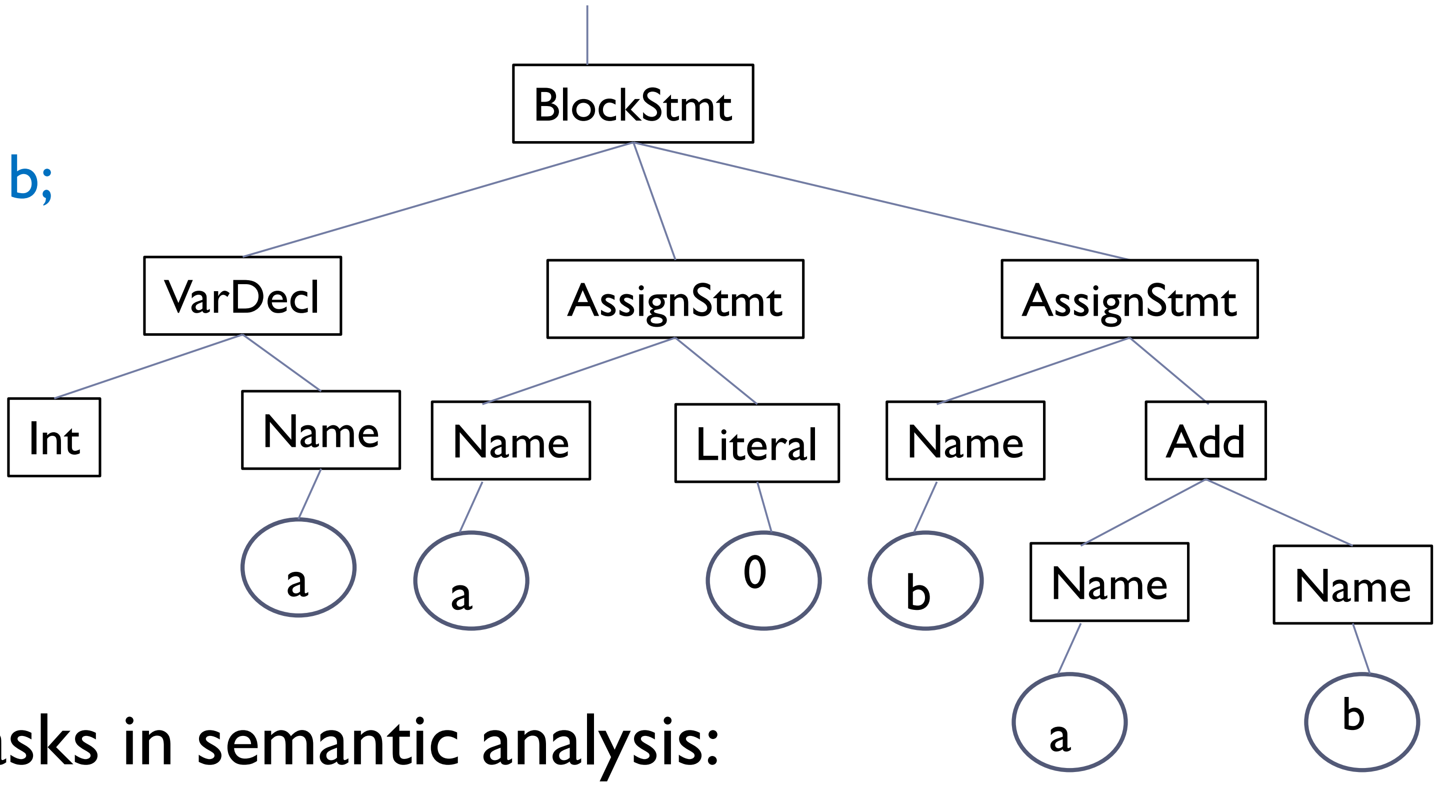
2) Given the CFG, the abstract grammar in slide 26 and an understanding of the semantic actions in slide 27, draw the AST for each of the following expressions.

a) (6)
b) (6 + 3)
c) 1 * 2 * 3
d) 1 * (2 * 3)

Semantic Analysis Using ASTs

- Example: A program in a very simple language has the AST shown

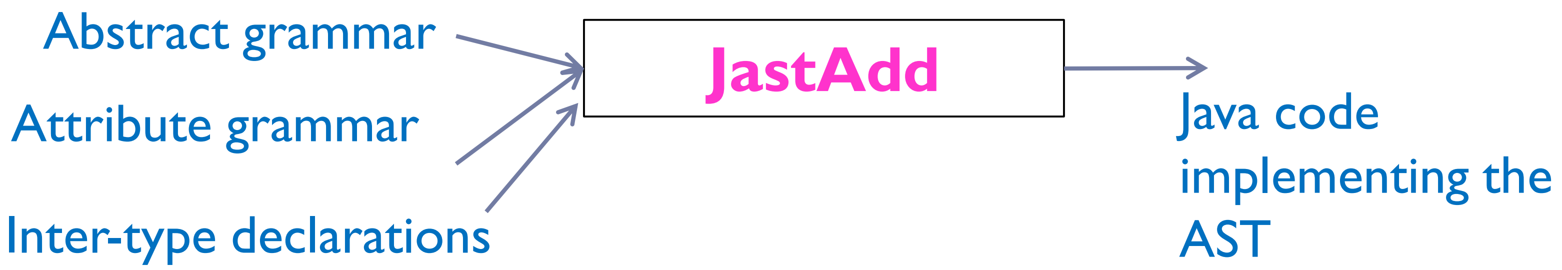
```
{ int a;
  a = 0;
  b = a + b;
}
```



- Two tasks in semantic analysis:
  1) Scope 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 classes need to have additional methods to compute information like where its declaration is or what its type is.
- For the compiler builder to define the methods on the AST classes: insert methods into AST classes by JastAdd
  1) Define methods in attribute grammar
  2) Write inter-type methods



Inter-Type Methods

- 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 of the class.
- A number of such inter-type methods that logically belong together can be collected together into an ***aspect*** and placed in a file with the extension .jrag.
- An aspect is like a module containing the inter-type declarations (methods and fields) addressing a certain concern for a number of AST classes.

Inter-Type Methods

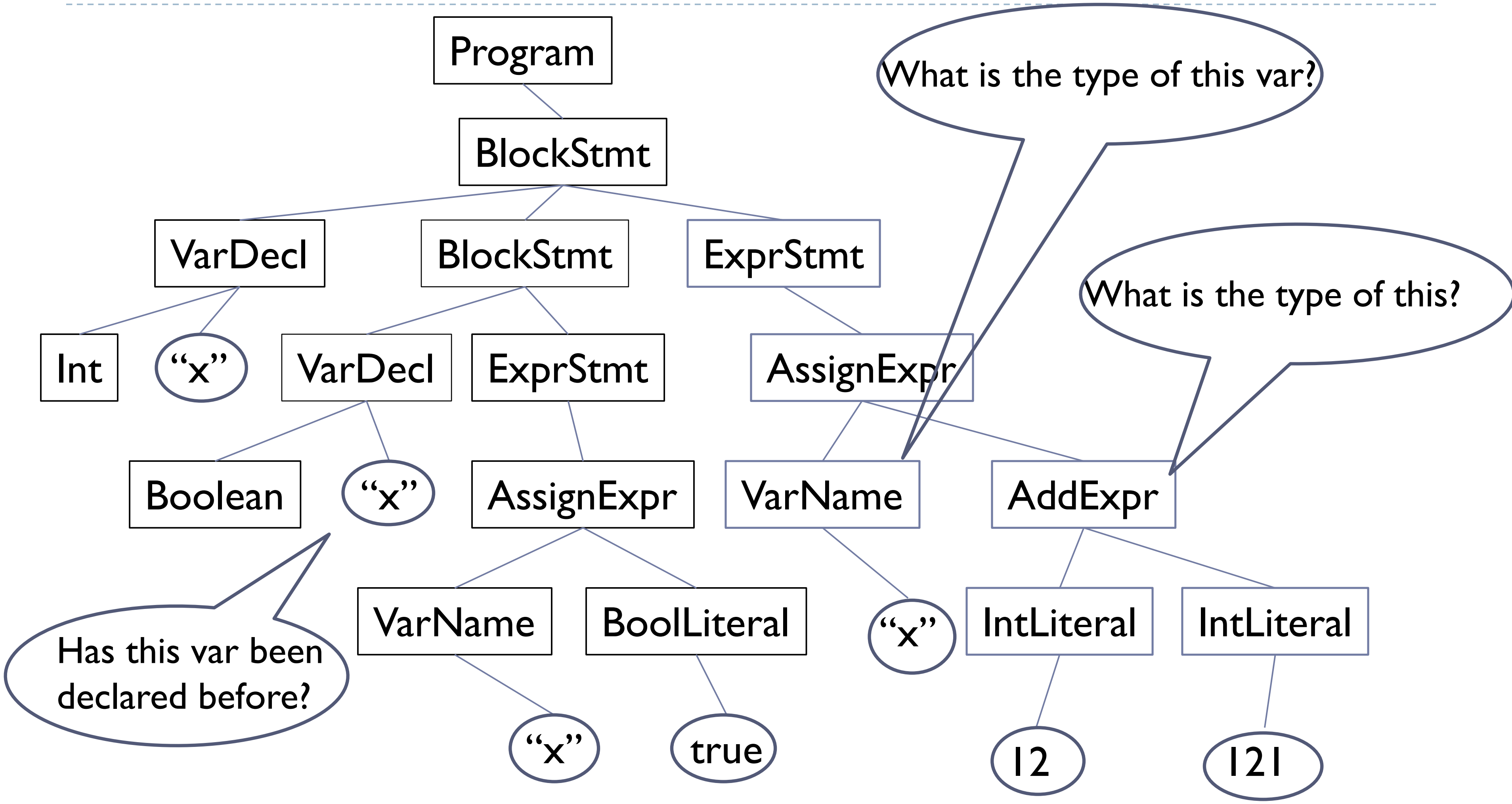
- When JastAdd generates the Java classes corresponding to node types, it can simply insert the definitions of the inter-type methods into the right AST class.
- The inter-type declarations are not a Java feature, but have to be translated into normal Java methods by JastAdd.
- Inter-type methods are discussed later in scope checking and type checking.

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 ancestor node (parent or higher), or even its sibling nodes.

For example:

Attribute Grammars



Attribute Grammars

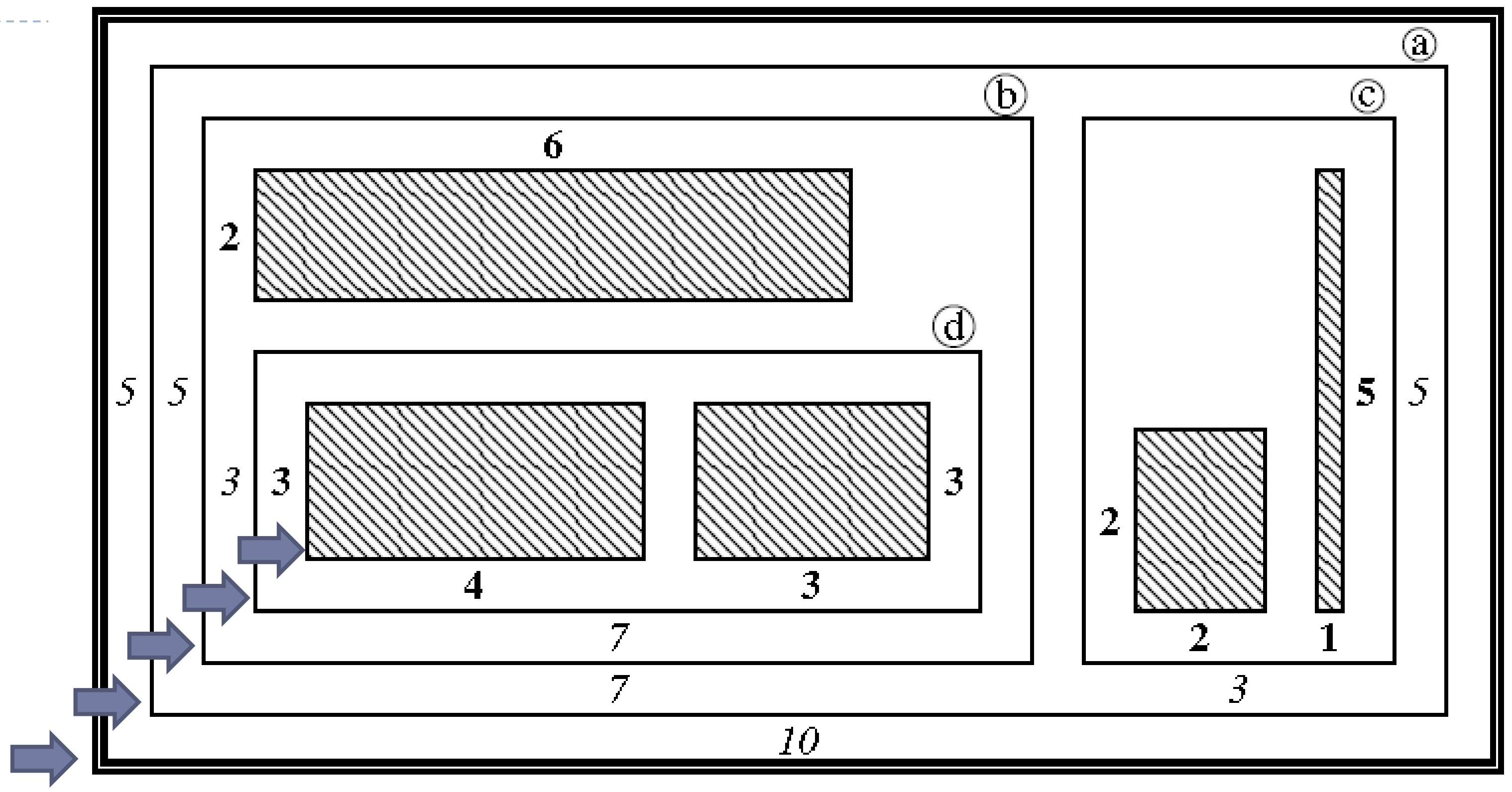
- Computations following these commonly occurring patterns can be described by ***attribute grammars***.
- An attribute grammar defines a set of attributes on top of an abstract grammar.
- An attribute is simply a function that computes a value for a certain kind of node.

Running Example

An abstract grammar for describing box scenarios:

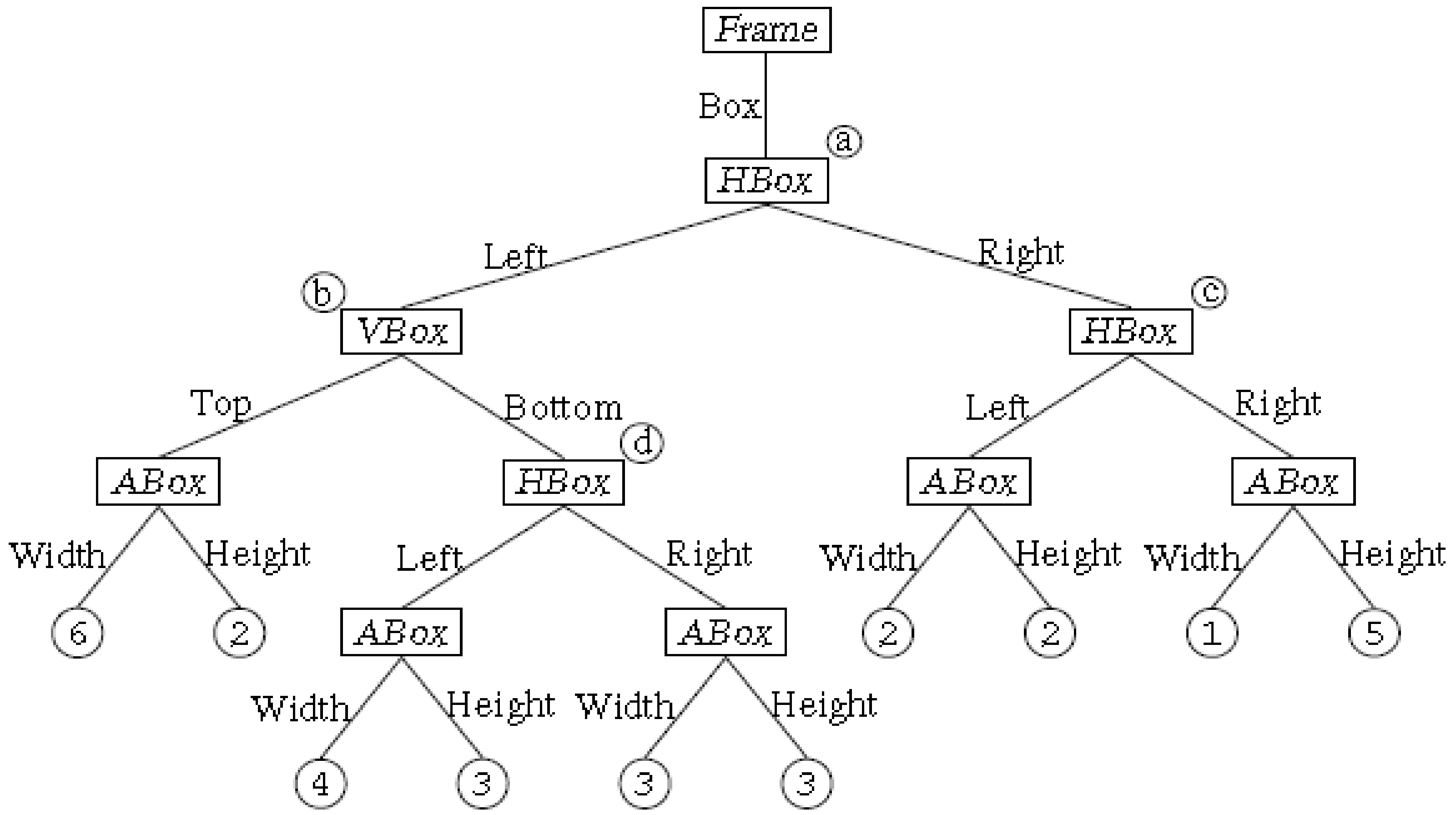
```
Frame ::= Box;
abstract Box;
HBox : Box ::= Left:Box  Right:Box;
VBox : Box ::= Top:Box  Bottom:Box;
ABox : Box ::= <Width:Integer>  <Height:Integer>;
```

An example of a box scenario



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

The AST Representation of the Scenario

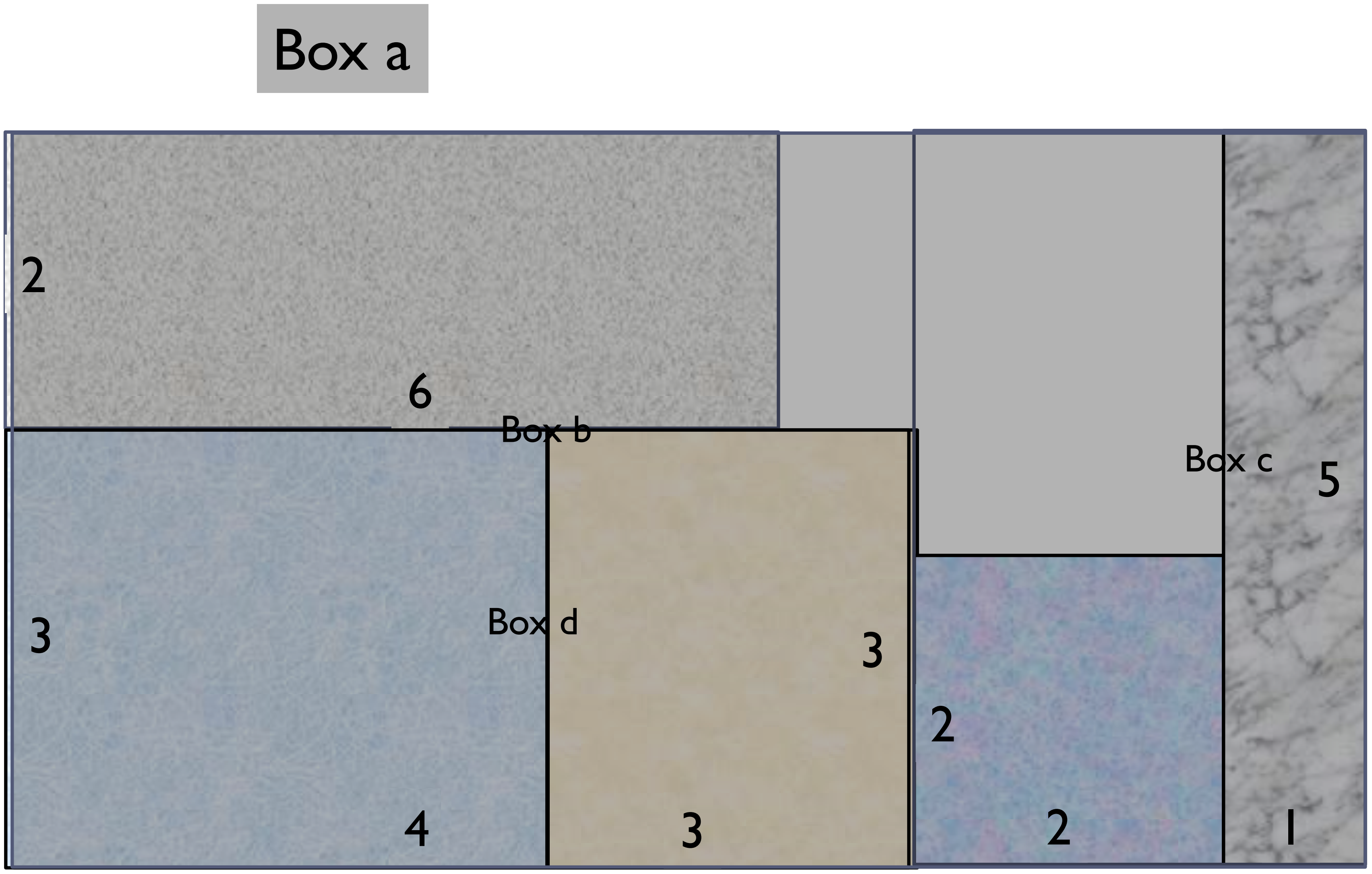


Synthesised Attributes

- A ***synthesised attribute a*** of some node ***n*** is an attribute whose value is computed based on the attribute values of the **children** of ***n*** in the AST and/or some other attributes of ***n***.
- Synthesised attribute is a way to pass information from lower level nodes to higher level nodes in an AST.
- In our abstract grammar, only atomic boxes have explicit widths and heights.

```
HBox : Box ::= Left:Box  Right:Box;
VBox : Box ::= Top:Box  Bottom:Box;
ABox : Box ::= <Width:Integer>  <Height:Integer>;
```

- The widths and heights of horizontal and vertical boxes are examples of ***synthesised attributes***.



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

Defining Synthesised Attributes

This is a synthesised attribute.

Attribute type

Attribute name

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, or code enclosed in { } without the '='.

- 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

```
// syn  int  Box.width();
public abstract class Box {
   ...
   abstract int width();
}
// eq  VBox.width() = Math.max(getTop().width(), getBottom().width());
public class VBox extends Box {
   ...
   int width() {
      return Math.max(getTop().width(),
                                         getBottom().width());
   }
}
```

JastAdd adds the method into the Box class.

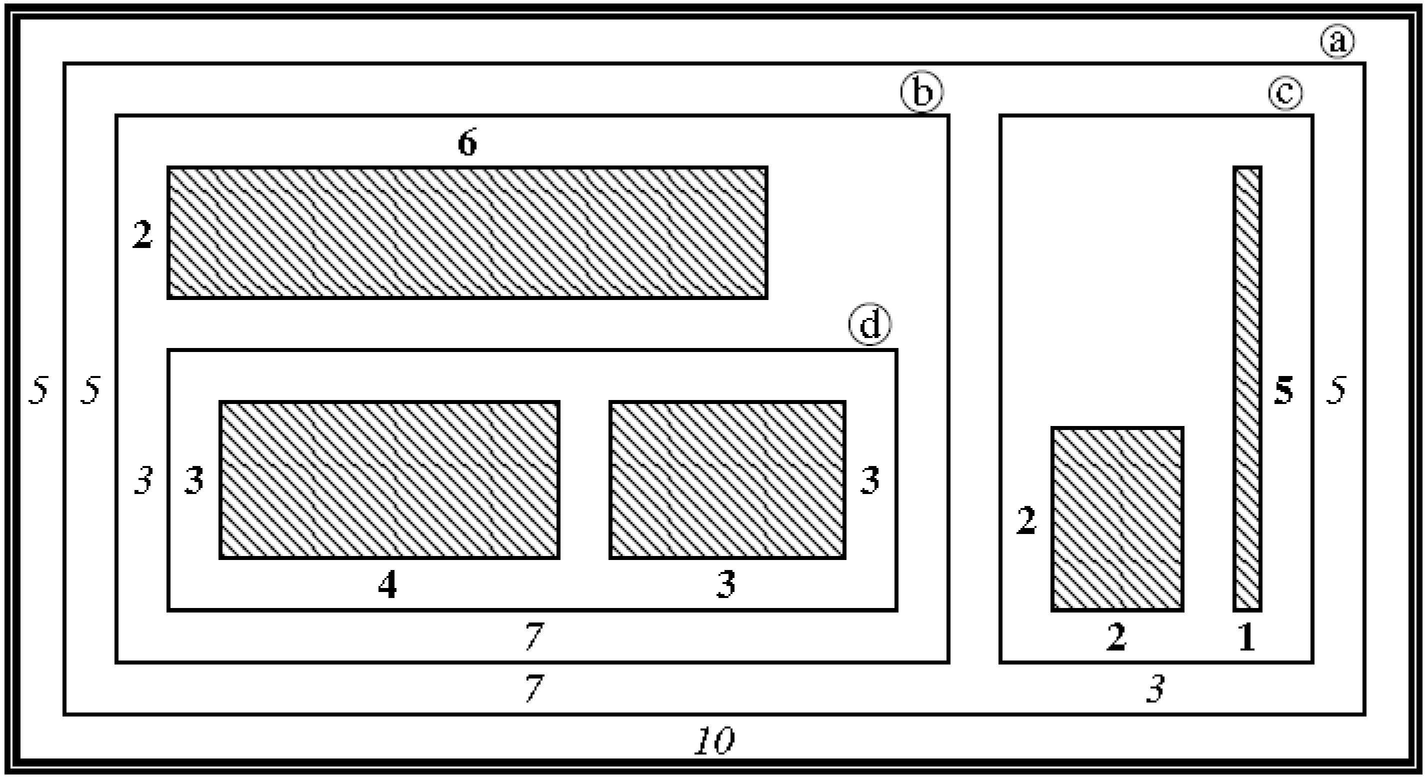
JastAdd adds the method into the VBox class.

Java Methods Generated for the Synthesised Attribute Width

```
// eq  HBox.width() = getLeft().width() + getRight().width();
public class HBox extends Box {
    ...
    int width() {
        return getLeft().width() + getRight().width();
    }
}
// eq  ABox.width() = getWidth();
public class ABox extends Box {
    ...
    int width() {
        return getWidth();
    }
}
```

Inherited Attributes

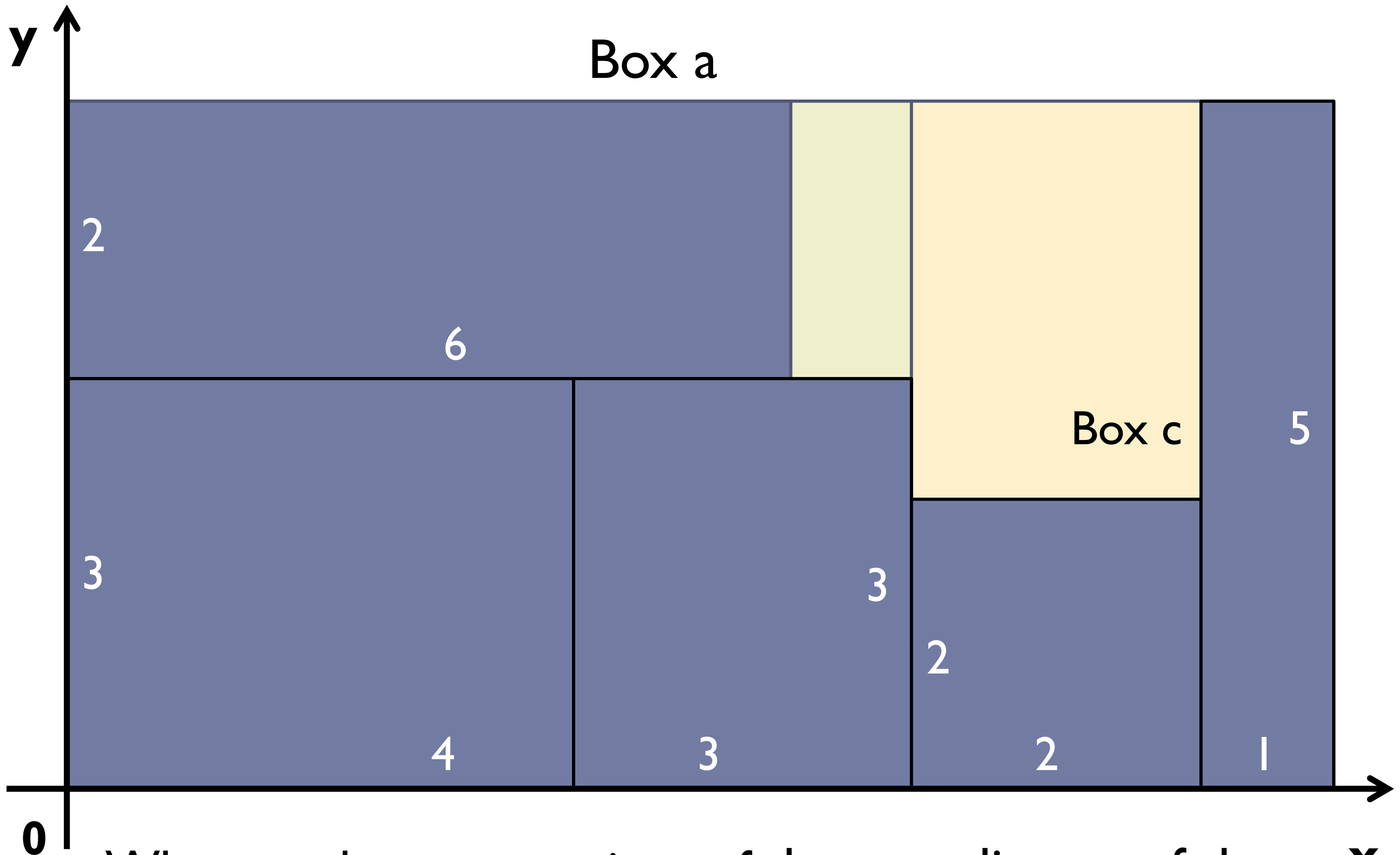
- Suppose we additionally want to find out the coordinates of every box's lower left corner.



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

Inherited Attributes

- The *x* and *y* coordinate values of the lower left corner are the attributes of concern.
- They are examples of ***inherited attributes***.
- An ***inherited attribute a*** of some node ***n*** is an attribute whose value is computed based on the attribute values of its ancestor node (parent or higher) and its **siblings** in the AST.
- Inherited attribute is a way to pass information from higher level nodes to lower level nodes in an AST.



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.

In this example, xpos is an inherited attribute declared in the super class.

```
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
```

The defining class <u>computes the attribute value</u> by using the Java code on the RHS. The result is passed to the child node's attribute.

Sibling of the right box

Attribute of the parent node of the right box

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().

  So Equations 4 and 5 can be replaced by a single equation:

  ```
  eq VBox.getChild().xpos() = xpos();
  ```
- This kind of equation is what is known as a ***copy equation*** where **the value of the inherited attribute of the child is simply defined to be the same as its value of the parent**; JastAdd allows to simply omit them.

Defining Inherited Attributes

For example, omitting the copy equations for ypos:

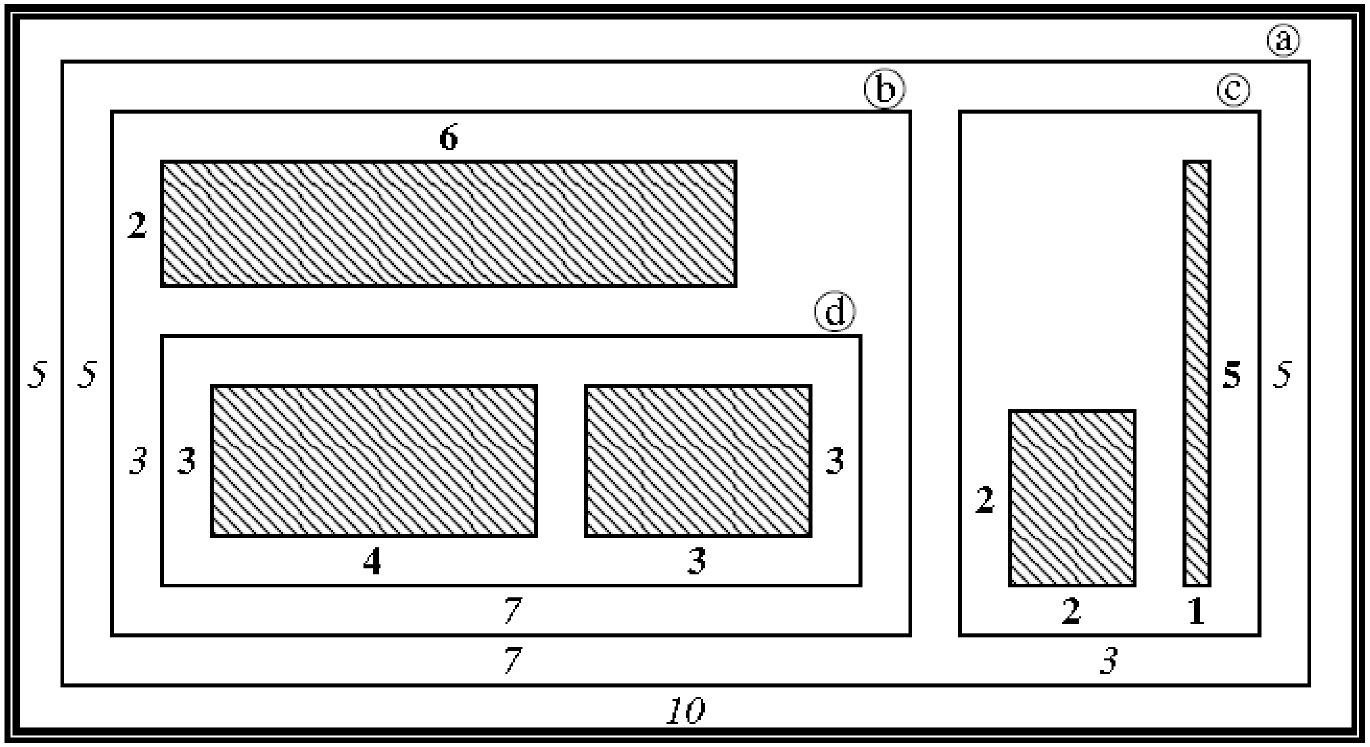
```
inh int Box.ypos();
eq  Frame.getBox().ypos() = 0;
eq  VBox.getTop().ypos() = ypos() + getBottom().height();
```

- 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



Lazy Attributes

- 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.

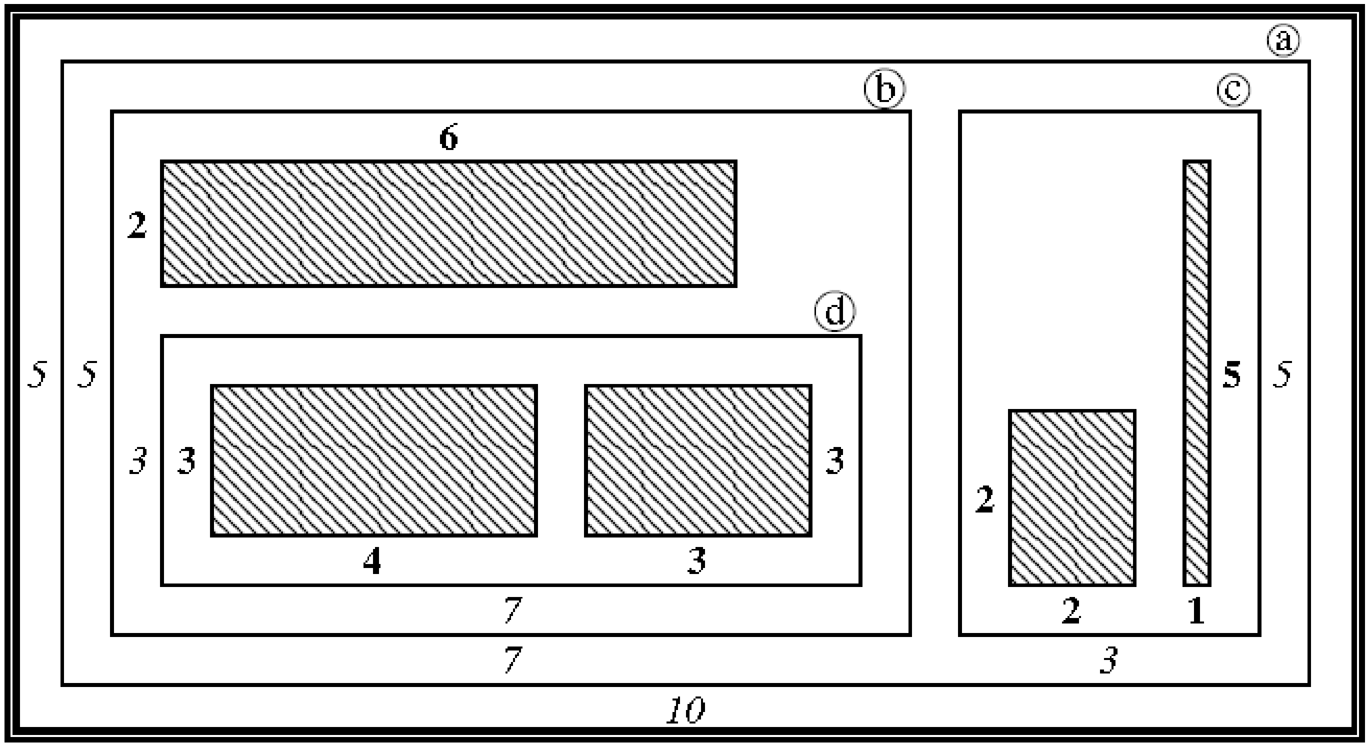
  E.g., **syn lazy int** Box.width();

  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



Reference Attributes

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:

```
 eq HBox.getLeft().leftNeighbour() = leftNeighbour();
 eq VBox.getChild().leftNeighbour() = leftNeighbour();
```

)

Parameterised Attributes

- Each of the attributes we have seen so far has a single value. E.g., width, xpos of node Box

  ```
  syn  int  Box.width();
  inh  int  Box.xpos();
  ```

- Some attribute value of an AST node depends on some parameters.
- A parameterised attribute will have one value for each possible combination of parameter values. For example, the lookupVar() attribute in slide 79.

Test Yourself 4.2

Given the CFG in slide 13 and the abstract grammar in slide 16, define in attribute grammar an attribute *value* for the node type *Expr* that returns the value of an expression. Examples: “2” has the value 2, “3+10” has the value 13, “2*(3+10)” has the value 26.

Name and Type Analysis

- The goal of name analysis is to determine, for every identifier appearing in the program, which declaration is that identifier's.
- Type analysis computes the types of composite expressions from the types of their components.
- Based on name analysis, scope checking (name checking) can be performed. For example, for Java, no two fields in the same class should have the same name.
- Based on type analysis, we can perform type checking.

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;

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 LHSExpr : Expr;
VarName : LHSExpr ::= <Name:String>;
ArrayIndex : LHSExpr ::= Base:Expr  Index:Expr;
AssignExpr : Expr ::= Left:LHSExpr  Right:Expr;
abstract BinaryExpr : Expr ::= Left:Expr  Right:Expr;
AddExpr : BinaryExpr;
MulExpr : BinaryExpr;
```

Example Language

```
abstract TypeName;
Int : TypeName;
Boolean : TypeName;
ArrayTypeName : TypeName ::= ElementType :TypeName;
```

- Note ArrayLiteral allows array literals like

  [1, 2, 3]   or [[1, 2], [3, 4], [5, 6]] or [[1, 2], [3, 4, 5], [6]]

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

- Note also there are three class hierarchies in the abstract grammar.

Example Program

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

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.

Scope of y

```
1 { int x;
2   int [][] y;
3   y = [ [ 1, 2 ], [ 3 ] ];
4   { // nested block
5       boolean x;
6       x = true;
7       if (x) {
8           y[0][1] = 23;
9       } else {
10          y[0][1] = 42;
11      }
12  }
13 }
```

Scope of x

```
1 { int x;
2   int [][] y;
3   y = [ [ 1, 2 ], [ 3 ] ];
4   { // nested block
5       boolean x;
6       x = true;
7       if (x) {
8           y[0][1] = 23;
9       } else {
10          y[0][1] = 42;
11      }
12  }
13 }
```

Scoping and Namespaces

- In our language, variables are the only named entities.
- In some languages like Java, there are five categories of program entities that have names: packages, types (i.e., classes and interfaces), methods, variables (including fields), and labels for statements.
- Scopes for packages, types, methods, variables, and labels for 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.

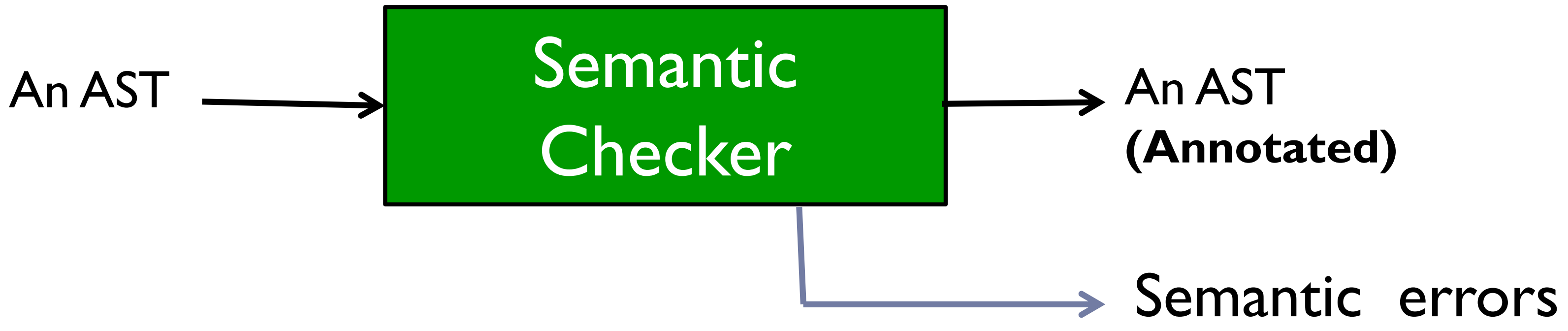
Example of Overlapping Scope

```
{   // this is not a program in our example language
    int biggest;
    int biggest(int x,  int y)
     {
        return Math.max(x, y);
     }
     biggest = biggest(100, 105);
}
```

The local variable biggest and the method name biggest are in separate name spaces

Scoping and Namespaces

- 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 names with information about the declaration they refer to.
- We will discuss a different approach: Annotate an AST with semantic information using attributes.



Basic Name Analysis

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

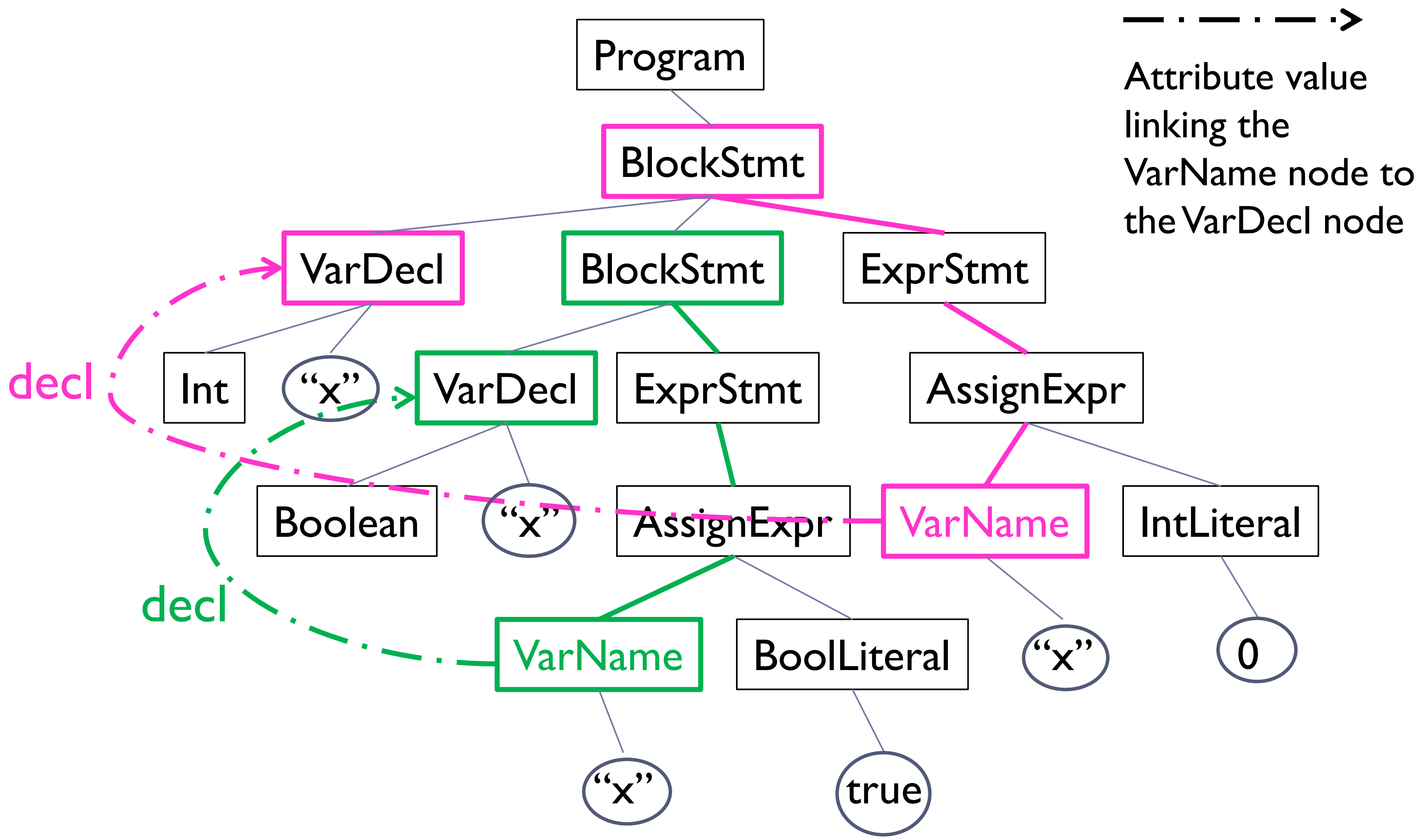
```
{   int  x;
    {     boolean  x;
          x = true;
    }
    x = 0;
}
```

Recall some AST classes in our Example Language

The complete abstract grammar is in slides 66-68

```
Program ::= BlockStmt;

abstract Stmt;

VarDecl : Stmt ::= TypeName  <Name:String>;
BlockStmt : Stmt ::= Stmt*;
ExprStmt : Stmt ::= Expr;
abstract Expr;

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

abstract LHSExpr : Expr;

VarName : LHSExpr ::= <Name:String>;
AssignExpr : Expr ::= Left:LHSExpr  Right:Expr;
```

Attribute decl on node type VarName



Implement the decl Attribute in JastAdd

- For a VarName node, we need to locate the VarDecl node from higher levels in the AST. Then we need to pass the reference to the VarDecl node down to the VarName node.
- We define an inherited attribute **lookupVar** for all Expr and Stmt node types of the AST tree. So VarName inherits this attribute.
- The attribute decl of a VarName just needs to get its value from the lookupVar attribute of the same VarName node. So the **decl** attribute is a synthesised attribute.

An example of a parameterised attribute introduced in Slide 63

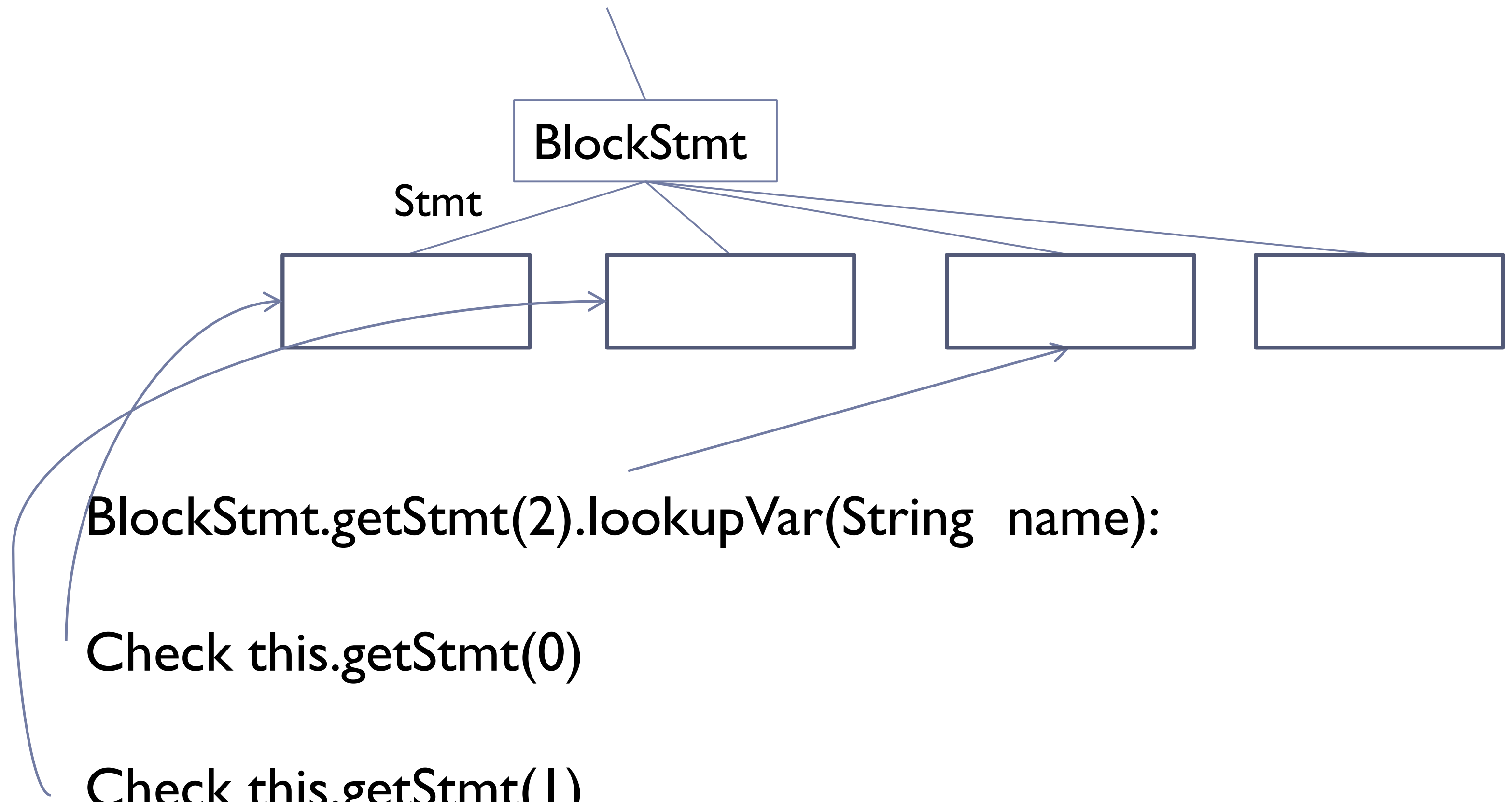
```
syn  VarDecl  VarName.decl() = lookupVar(getName());
inh  lazy  VarDecl  Expr.lookupVar(String name);
inh  lazy  VarDecl  Stmt.lookupVar(String name);
eq  Program.getChild().lookupVar(String name) = null;
eq  BlockStmt.getStmt(int i).lookupVar(String name) {
      for ( int  j = 0;  j < i;  ++j) {
          Stmt stmt = this.getStmt(j);
          if  (stmt   instanceof  VarDecl) {
              VarDecl  decl = (VarDecl) stmt;
              if  (decl.getName().equals(name))
                  return decl;
          }
      }
      return this.lookupVar(name);                                 }
```

BlockStmt node executes the code and the result passed to the ith child node.

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

Lookup in the parent node

Example



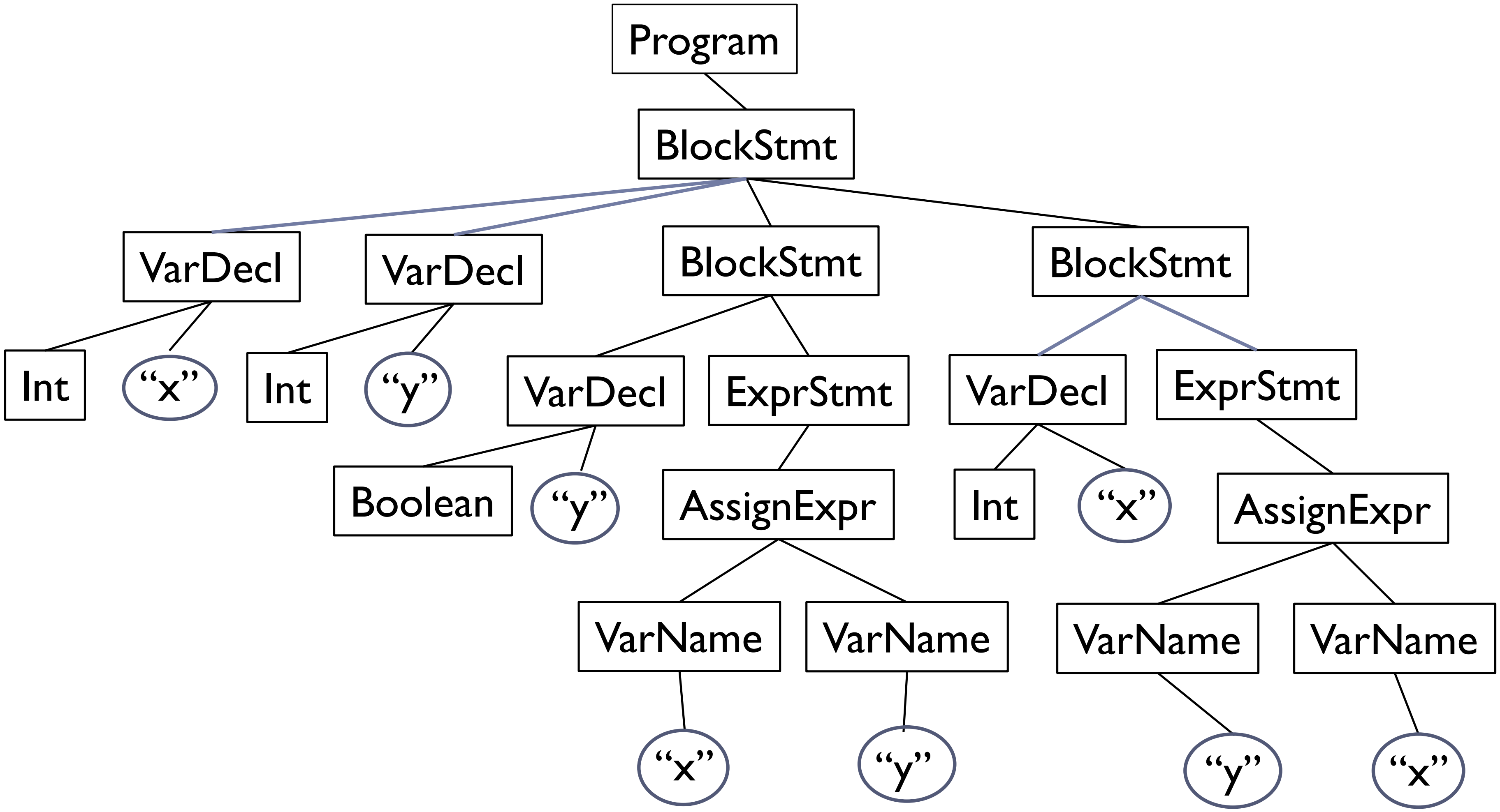
BlockStmt.getStmt(2).lookupVar(String name):

Check this.getStmt(0)

Check this.getStmt(1)

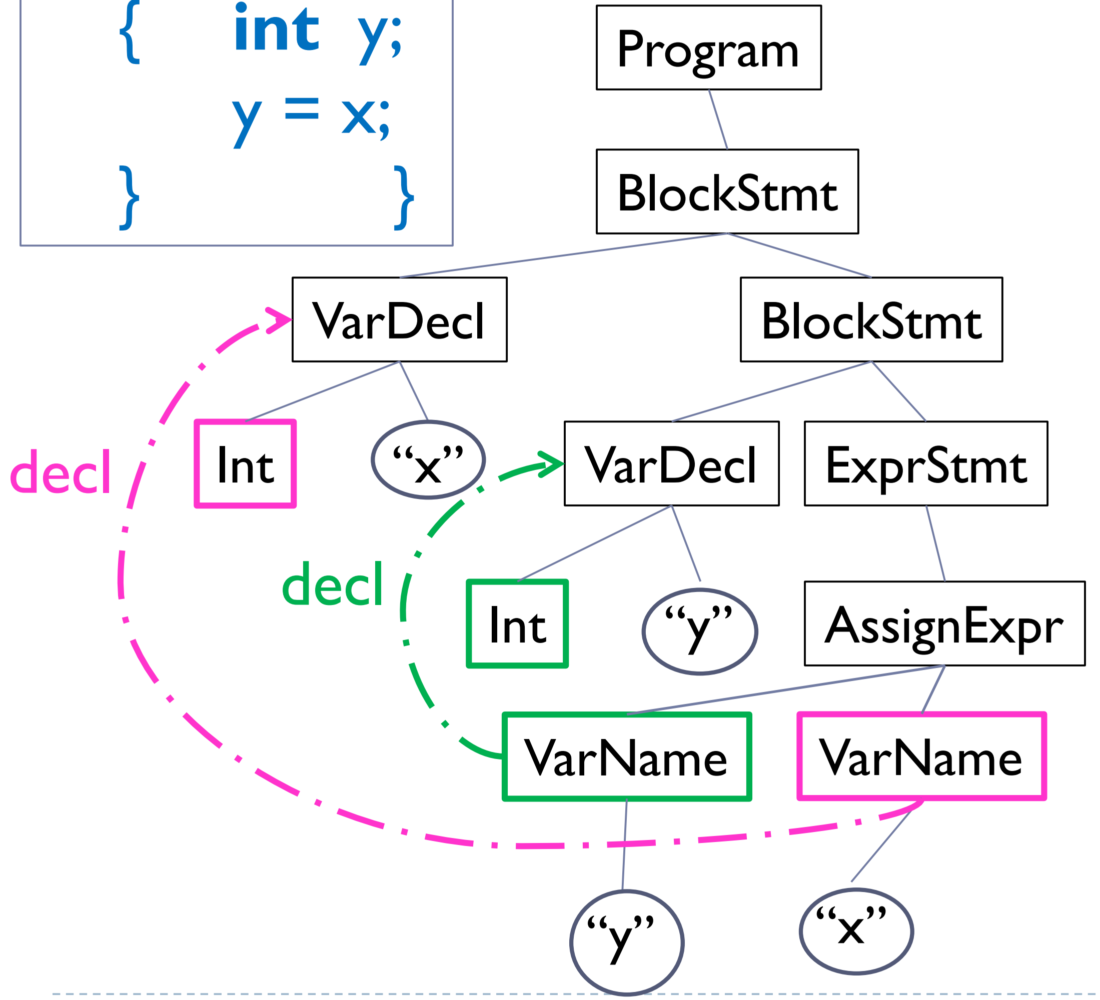
Test Yourself 4.3

Which BlockStmt node(s) will take part in the search for the declaration of each of the variables in the two assignment statements?



Basic Type Analysis

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



Type analysis is to determine the type of an expression/method so that we can do type checking.

Consider type checking for AssignExpr. Can we do it this way:

getLeft().decl().getTypeName()
==
getRight().decl().getTypeName()

The condition will return false!

Basic Type Analysis

abstract Expr;

- We define a synthesised attribute ***type*** on the node type *Expr* that computes the type of an expression node.
- We declare **syn** TypeDescriptor Expr.type();
- All the subclasses of Expr will inherit this attribute. E.g. VarName, IntLiteral.
- We use a new node type called ***TypeDescriptor*** to annotate the type of an expression node. Type descriptors can be defined by **abstract grammar** in the **.ast** file. E.g.,

Basic Type Analysis

Recall some AST classes:

abstract TypeName;

Int : TypeName;

Boolean : TypeName;

ArrayTypeName : TypeName ::= ElementType :TypeName;

The new node types defined for these type names:

abstract TypeDescriptor;

IntType :TypeDescriptor;

BooleanType :TypeDescriptor;

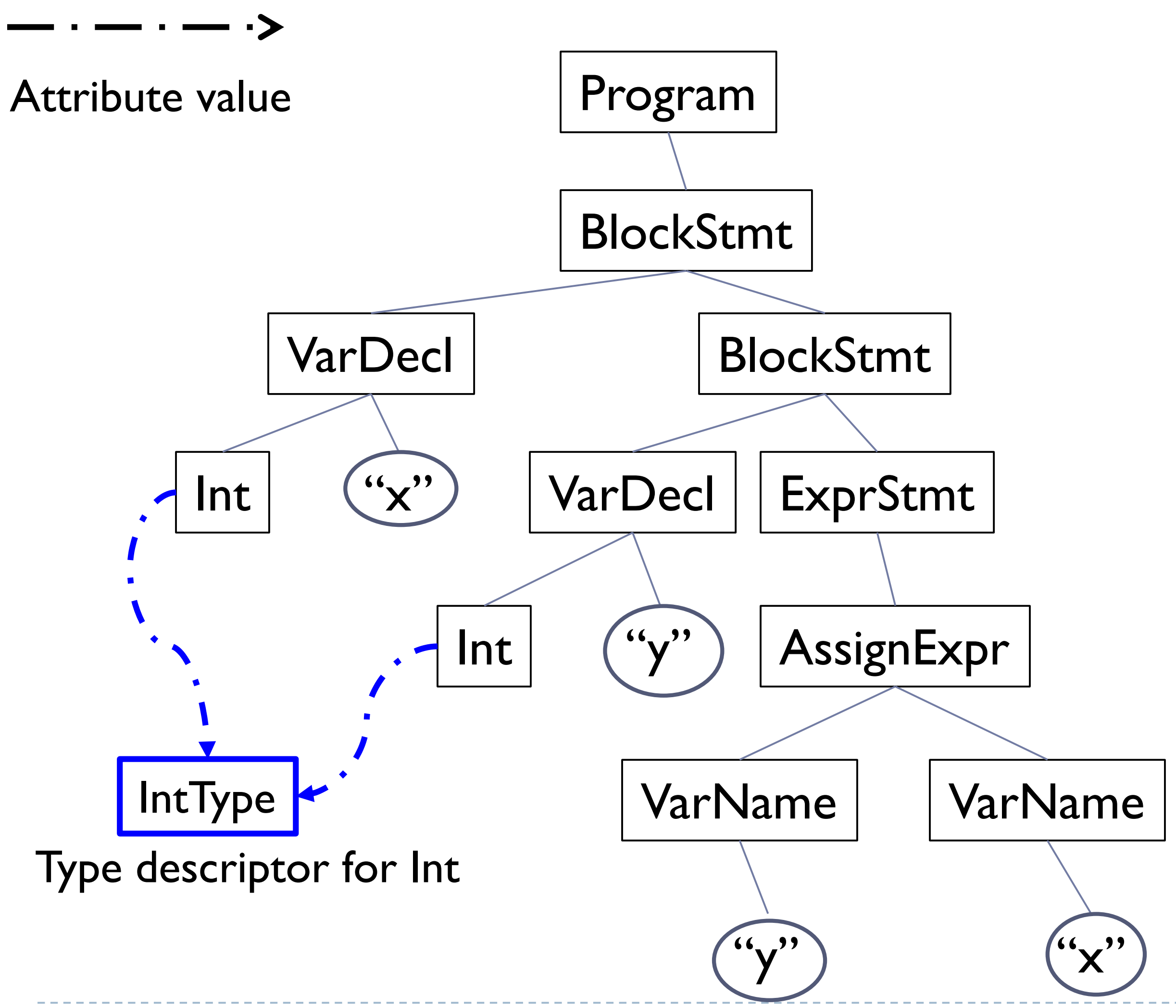
ArrayType :TypeDescriptor ::= ElementType:TypeDescriptor;

The Type Descriptor of a Type Name

- Each TypeName node will be linked to a type descriptor.
- For "Int" and "Boolean", each has a single instance of the type descriptor, an IntType or a BooleanType.
- We let the super class TypeDescriptor provide two static fields to reference the single instances of IntType and BooleanType. These are inter-type field declarations made in an aspect in **.jrag** file.

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

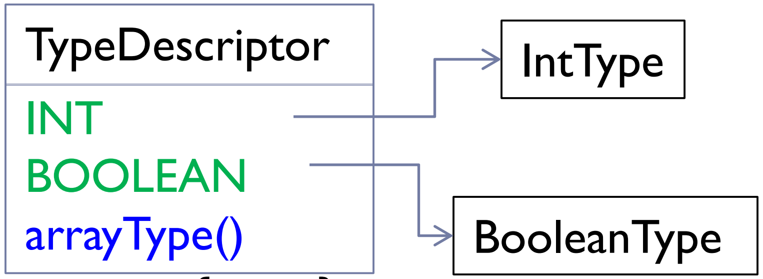
The Type Descriptor of a Type Name



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

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;     }
….}
```



A note about arrayType()

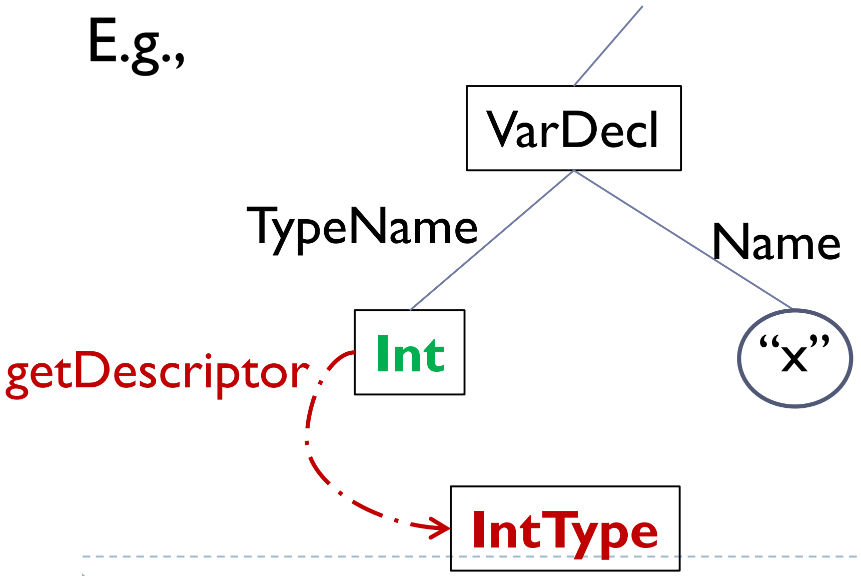
```
public abstract class TypeDescriptor {
   public static IntType INT = new IntType();
   public static BooleanType BOOLEAN = new BooleanType();
   public ArrayType  arrayType() {
      return   new  ArrayType(this);   }
   ….     }
   public class IntType extends TypeDescriptor  { ….}
   public class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

A method in super class TypeDescriptor.
It creates an ArrayType node with its ElementType to be the caller's node type

Linking TypeName Nodes to Type Descriptors

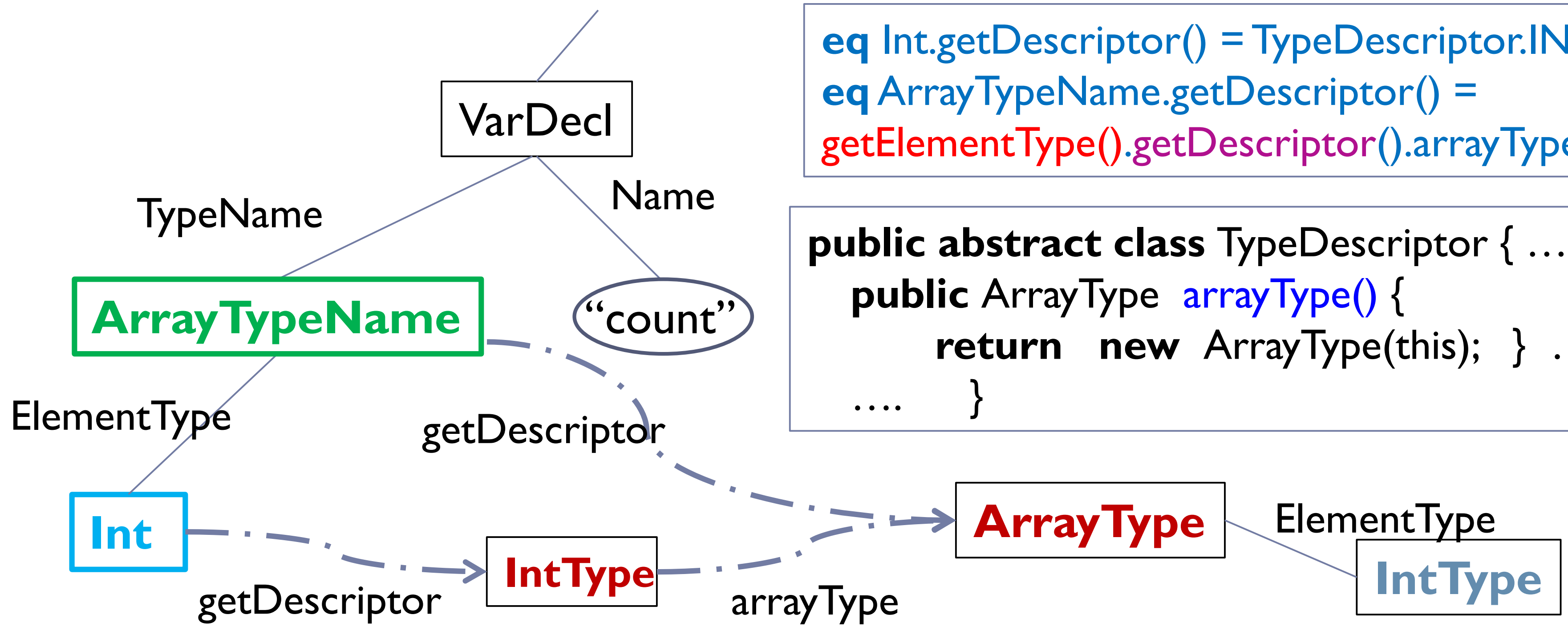
```
syn lazy TypeDescriptor TypeName.getDescriptor();
eq Int.getDescriptor() = TypeDescriptor.INT;
eq Boolean.getDescriptor() = TypeDescriptor.BOOLEAN;
eq ArrayTypeName.getDescriptor() =
                    getElementType().getDescriptor().arrayType();
```

E.g.,



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

Linking TypeName Nodes to Type Descriptors



```
eq Int.getDescriptor() = TypeDescriptor.INT;
eq ArrayTypeName.getDescriptor() =
getElementType().getDescriptor().arrayType();
```

```
public abstract class TypeDescriptor { …
   public ArrayType  arrayType() {
         return   new  ArrayType(this);  }  …
   ….     }
```

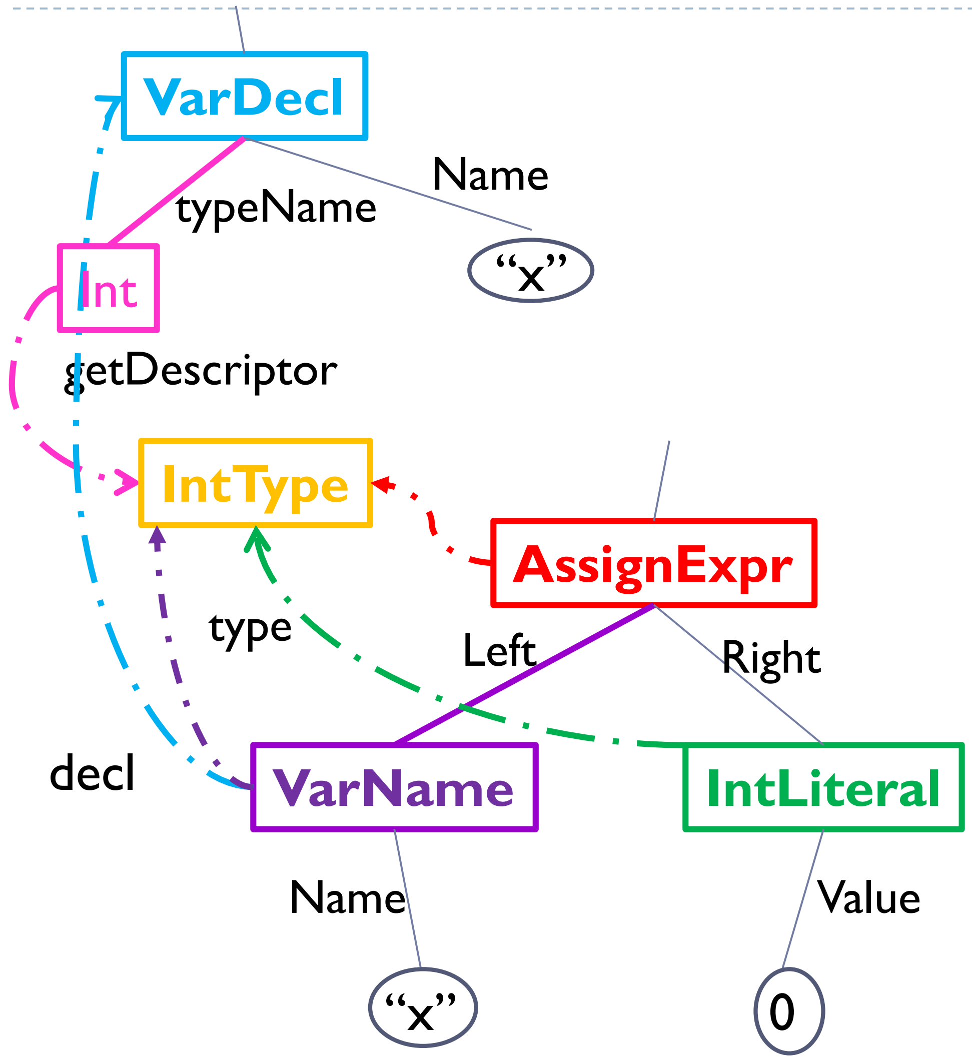
Firstly, getDescriptor() for node **ArrayTypeName** will call
getElementType() which returns **Int**.
Secondly, **Int**.getDescriptor() returns a type descriptor IntType.
Finally, IntType.arrayType() returns **ArrayType** with child **IntType**.

The Type of an Expression

- Every expression node needs to compute its type.

```
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() = getLeft().type();
eq BinaryExpr.type() = TypeDescriptor.INT;
```

The Type of an IntLiteral/VarName/AssignExpr



IntLiteral.type() = TypeDescriptor.INT

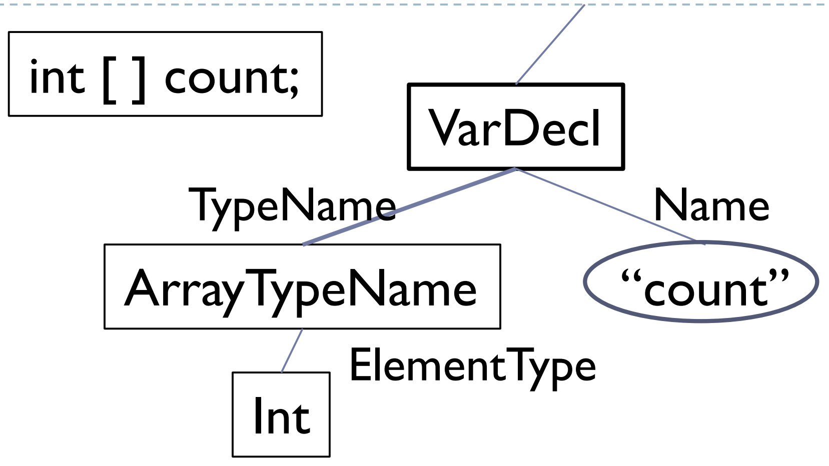
VarName.type() = decl(). getTypeName().getDescriptor()

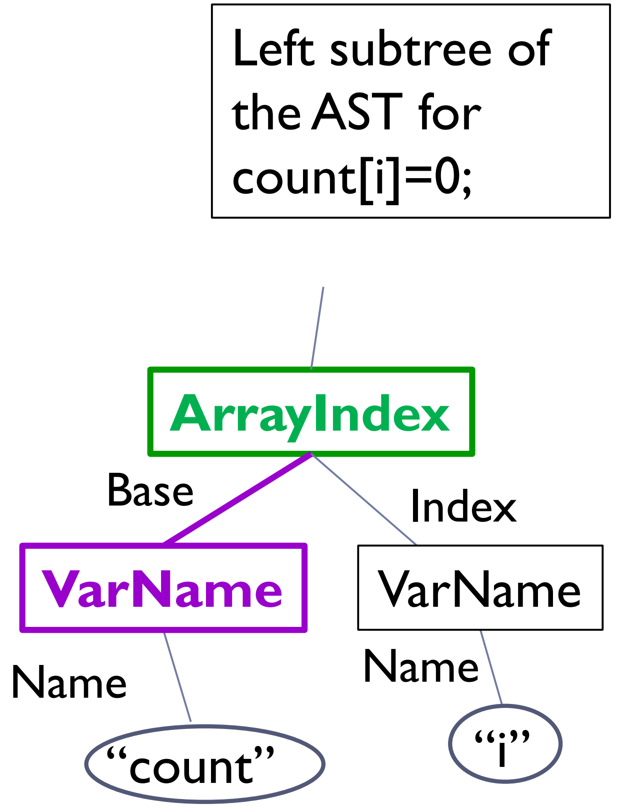
1. decl() returns the reference to **VarDecl**;
2. VarDecl.getTypeName() returns TypeName **Int**;
3. Int.getDescriptor() = TypeDescriptor.INT

AssignExpr.type() = getLeft().type() which is VarName.type()

The Type of an ArrayIndex (count[i])

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

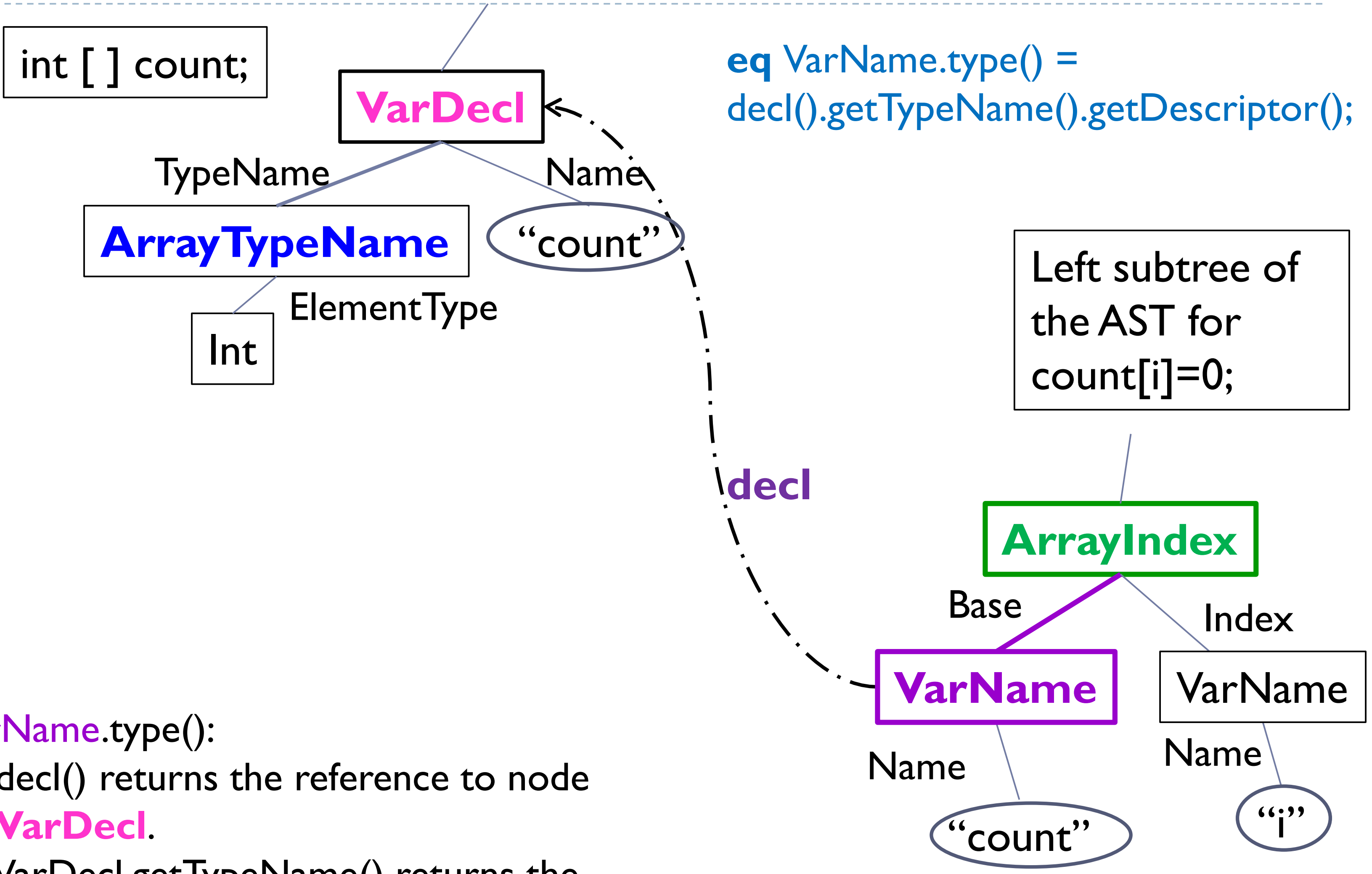




1. getBase() returns the reference to node **VarName**;

The Type of an ArrayIndex (count[i])

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



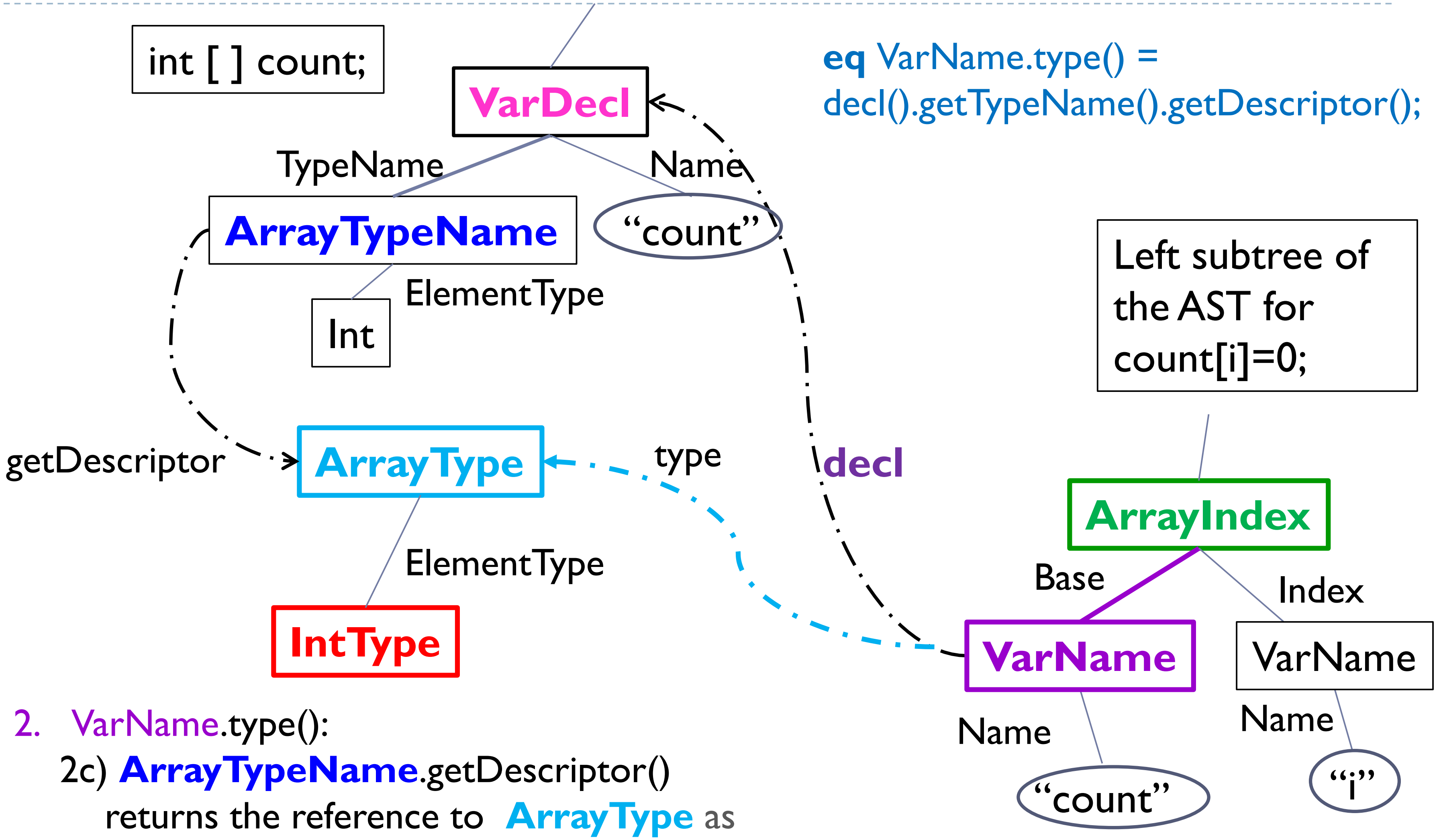
eq VarName.type() = decl().getTypeName().getDescriptor();

2. VarName.type():
   2a) decl() returns the reference to node **VarDecl**.
   2b) VarDecl.getTypeName() returns the reference to **ArrayTypeName**.

back

The Type of an ArrayIndex (count[i])

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



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

Left subtree of the AST for count[i]=0;

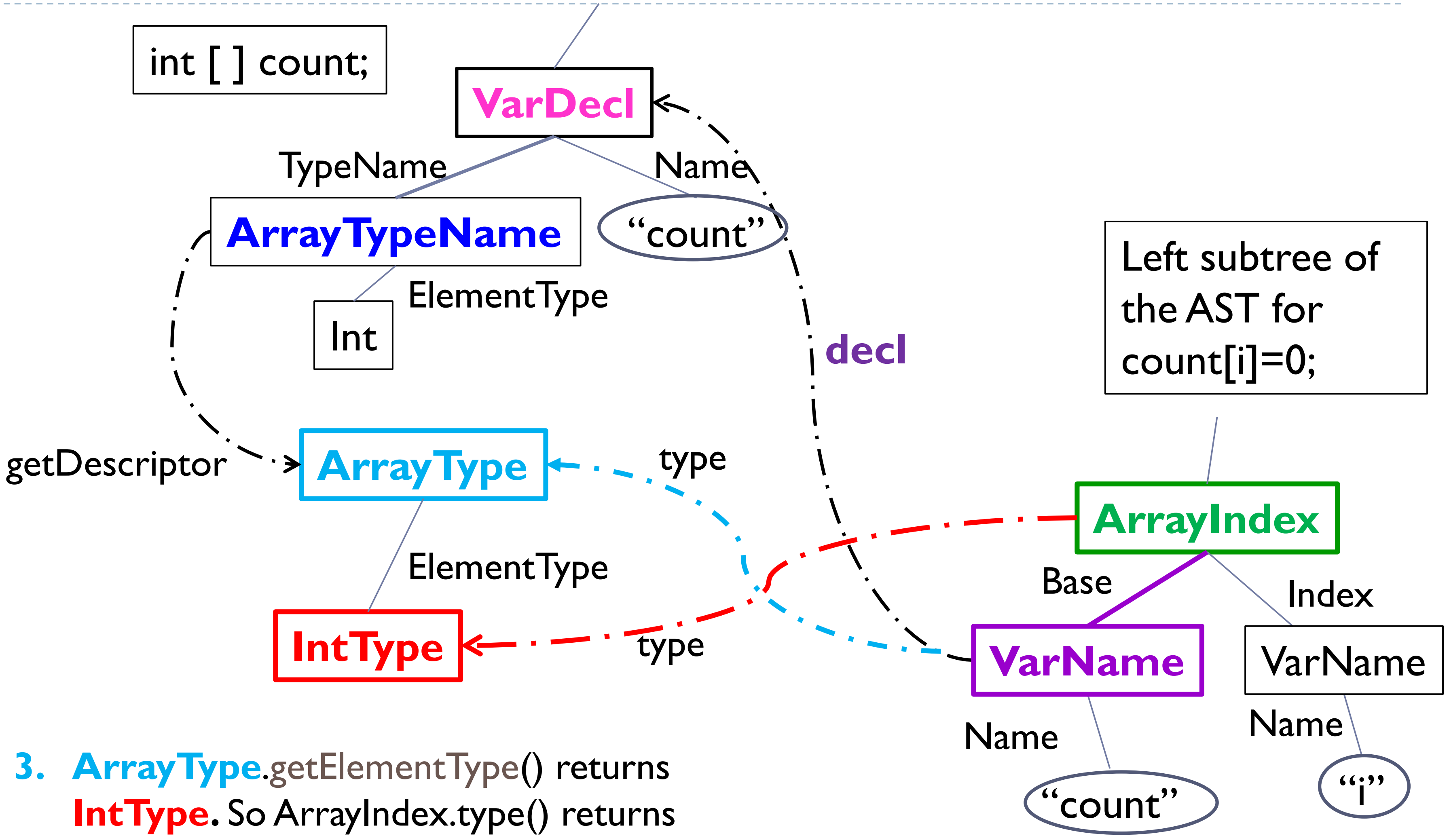
2. VarName.type():

   2c) **ArrayTypeName**.getDescriptor() returns the reference to **ArrayType** as shown in slide 90. So VarName.type() returns **ArrayType**

back

The Type of an ArrayIndex (count[i])

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



3. **ArrayType**.getElementType() returns **IntType**. So ArrayIndex.type() returns **IntType**.

back

Summary of evaluation steps (slides 93-96)

1. **ArrayIndex.type()**: getBase() returns the reference to node **VarName**;
2. VarName.type(): goes through the process according to

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

   a) Decl() returns the reference to node VarDecl.
   b) VarDecl.getTypeName() returns the reference to ArrayTypeName.
   c) ArrayTypeName.getDescriptor() returns the reference to **ArrayType** as shown in slide 90.

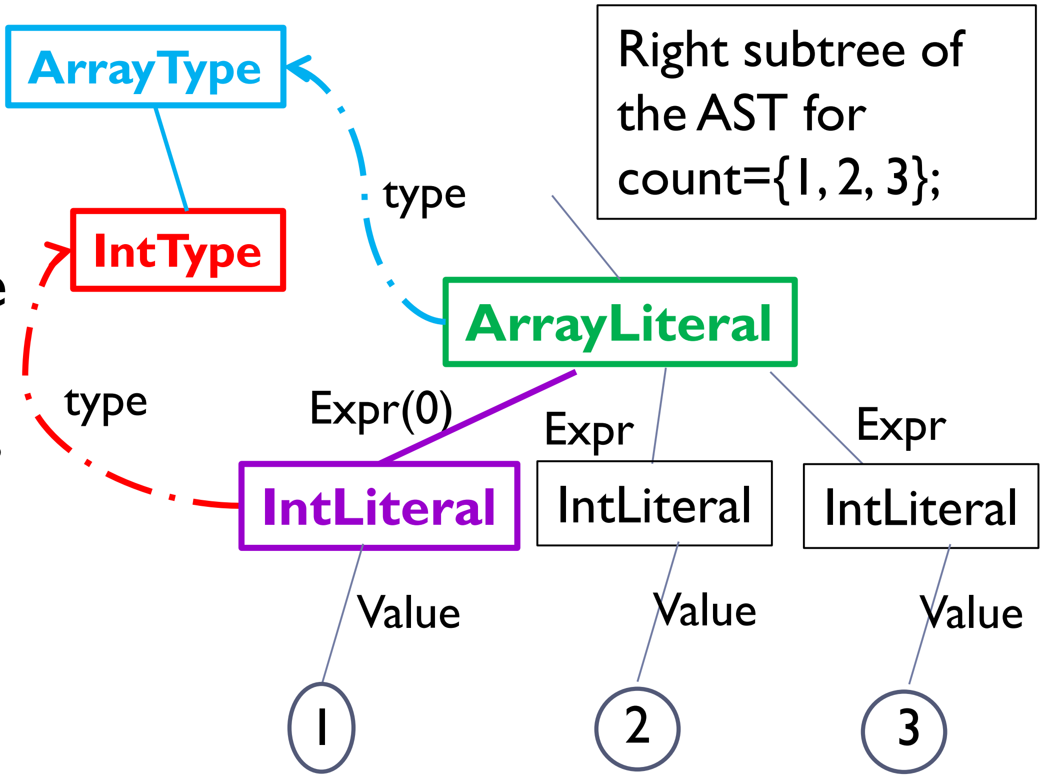
3. ArrayType.getElementType() returns **IntType**.

```
eq ArrayIndex.type() =
            ((ArrayType)getBase().type()).getElementType();
eq VarName.type() = decl().getTypeName().getDescriptor();
```

The Type of an ArrayLiteral ({1, 2, 3})

ArrayLiteral.type() :

1. getExpr(0) returns the reference to the first **IntLiteral** node;
2. IntLiteral.type() returns the type descriptor **IntType**;
3. IntType.arrayType() returns a type descriptor **ArrayType** with child **IntType** (illustrated in slide 90).

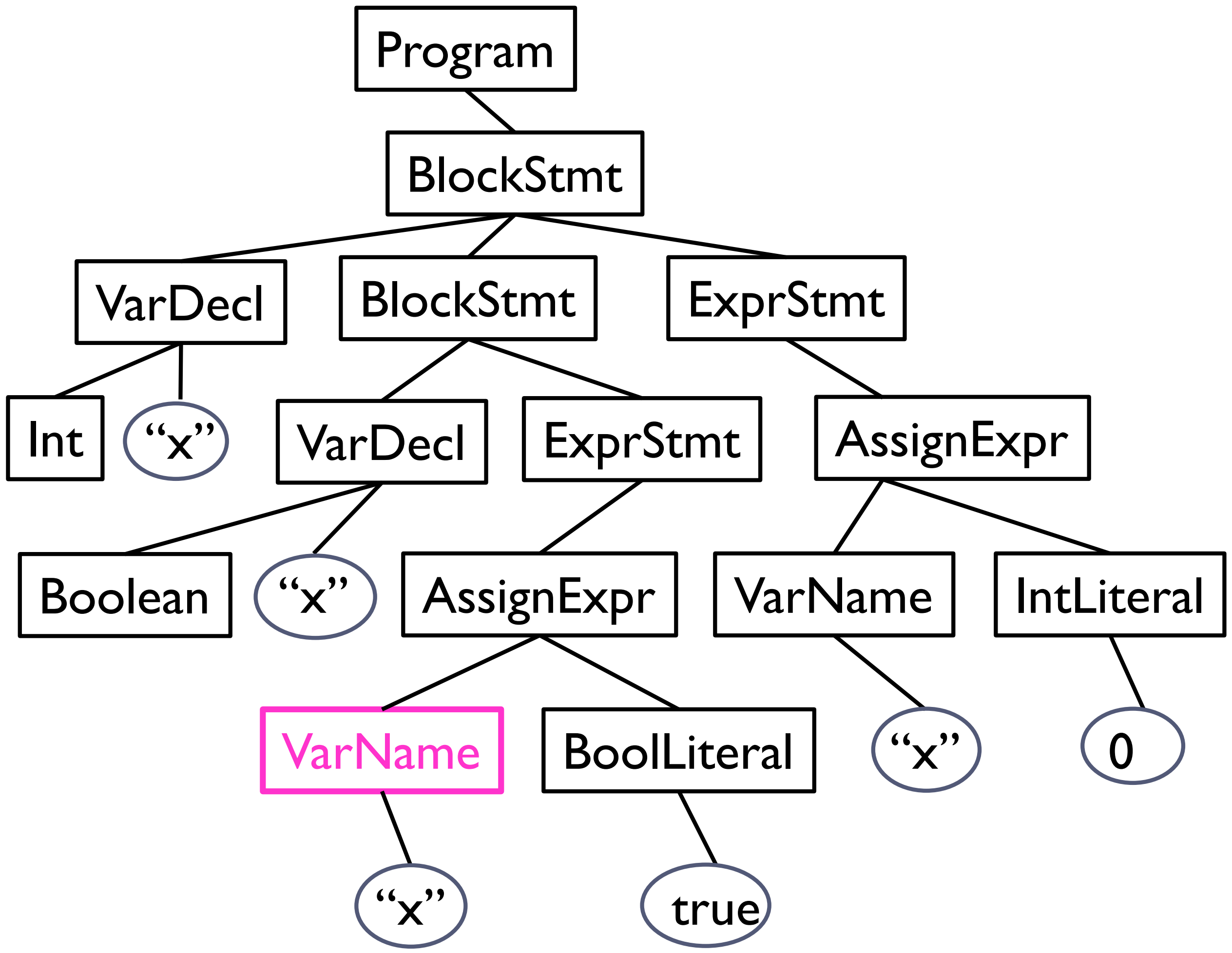


eq ArrayLiteral.type() = getExpr(0).type().arrayType();
eq IntLiteral.type() = TypeDescriptor.INT;

- Note that none of these equations perform any error checking (this is type analysis, not type checking). E.g.,
  - BinExpr.type() simply returns TypeDescriptor.INT without checking the operands.
  - AssignExpr.type() does not check whether the LHS and RHS are type compatible.
- These checks are done by separate type 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 ArrayTypeName. E.g. count[i]
  - VarName.type() does not check whether decl() returns null. (This error will be detected in name checking.)

Test Yourself 4.4

When VarName.type() is computed for the pink node, what will happen to the AST, and what is the result?



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.
- Another feature of object-oriented languages that name analysis needs to handle is inheritance: A class has its own fields, together with fields inherited from its super class.
- 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.
- 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.
- 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 and link a type name to its type descriptor.

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, operands and operators should be compactible, etc.
- Semantic error checking is implemented by defining a method ***check*** for various nodes in an AST.

Scope Checking for our Simple Language

- Scope checking is performed by the namecheck() methods on node types VarDecl and VarName.
- For VarName, we simply need to ensure that a declaration of that name is in scope.
- For VarDecl, we want to check that there isn't another variable of the same name declared within the same block.
- To make sure all VarName and VarDecl nodes are checked, we traverse the AST systematically starting from the root of the AST.

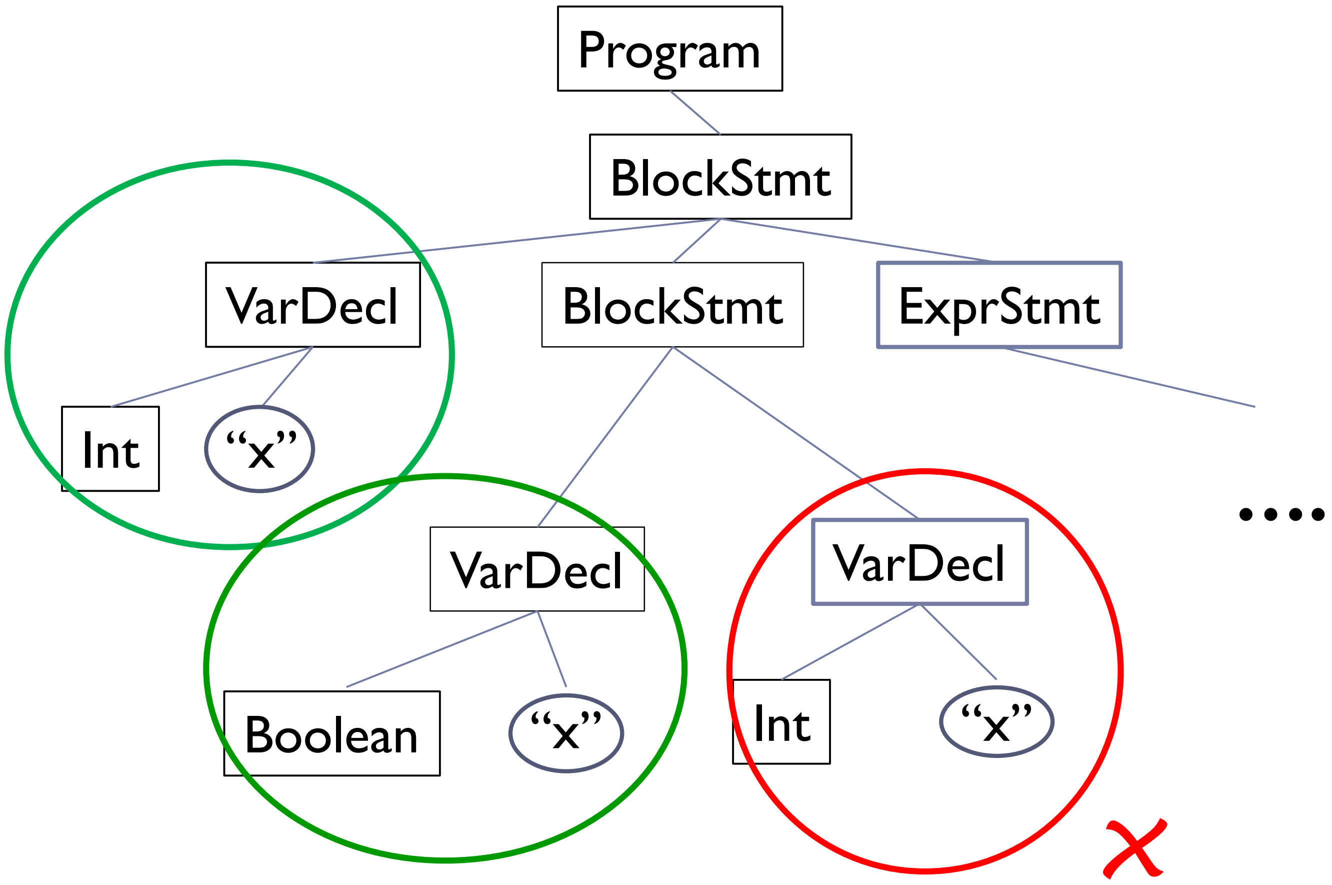
Scope Checking for our Simple Language (inter-type methods in .jrag file)

```
// Program ::= BlockStmt;
public void Program.namecheck() {
    getBlockStmt().namecheck();          }


// BlockStmt : Stmt ::= Stmt*;
public void BlockStmt.namecheck() {
    for (Stmt stmt : getStmts())      stmt.namecheck();      }


// WhileStmt : Stmt ::= Cond:Expr Body:Stmt;
public void WhileStmt.namecheck() {
     getCond().namecheck();      getBody.namecheck();      }
```

Scope Checking for our Simple Language (inter-type methods in .jrag file)

```
// VarDecl : Stmt ::= TypeName  <Name:String>;
public void VarDecl.namecheck() {
    VarDecl conflicting = lookupVar(getName());
    if (conflicting != null && conflicting.getParent() ==
                                                this.getParent())
        error("duplicate local variable with name“ +
                                                    getName());
}
// VarName : LHSExpr ::= <Name:String>;
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)

- In the error checking for statements, we start from the root node of the AST and recursively check their child statements and expressions.
- We do the type checking for our example language to show how it is done.

```
// Program ::= BlockStmt;
public void Program.typecheck() {
    getBlockStmt().typecheck();        }
// BlockStmt : Stmt ::= Stmt*;
public void BlockStmt.typecheck() {
    for (Stmt stmt : getStmts())      stmt.typecheck();      }
```

Type Checking for our Simple Language (inter-type methods in .jrag file)

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

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

// ExprStmt : Stmt ::= Expr;
public void ExprStmt.typecheck() {
    getExpr().typecheck();
}
```

Type Checking for our Simple Language (inter-type methods in .jrag file)

```
// IntLiteral : Expr ::= <Value:Integer>;
public void IntLiteral.typecheck() {   }
// typecheck() for BooleanLiteral and VarName will also have
// nothing to do

// AssignExpr : Expr ::= Left:LHSExpr  Right:Expr;
public void AssignExpr.typecheck() {
    getLeft().typecheck();
    getRight().typecheck();
    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)

```
// abstract BinaryExpr : Expr ::= Left:Expr  Right:Expr;
public void BinaryExpr.typecheck() {
    getLeft().typecheck();
    getRight().typecheck();
    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)

```
// ArrayLiteral : Expr ::= Expr*;
public void ArrayLiteral.typecheck() {
    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  !EqualType(getExpr(i).type(), tp)
                error("inconsistent types");
    }
}
```

Type Checking for our Simple Language (inter-type methods in .jrag file)

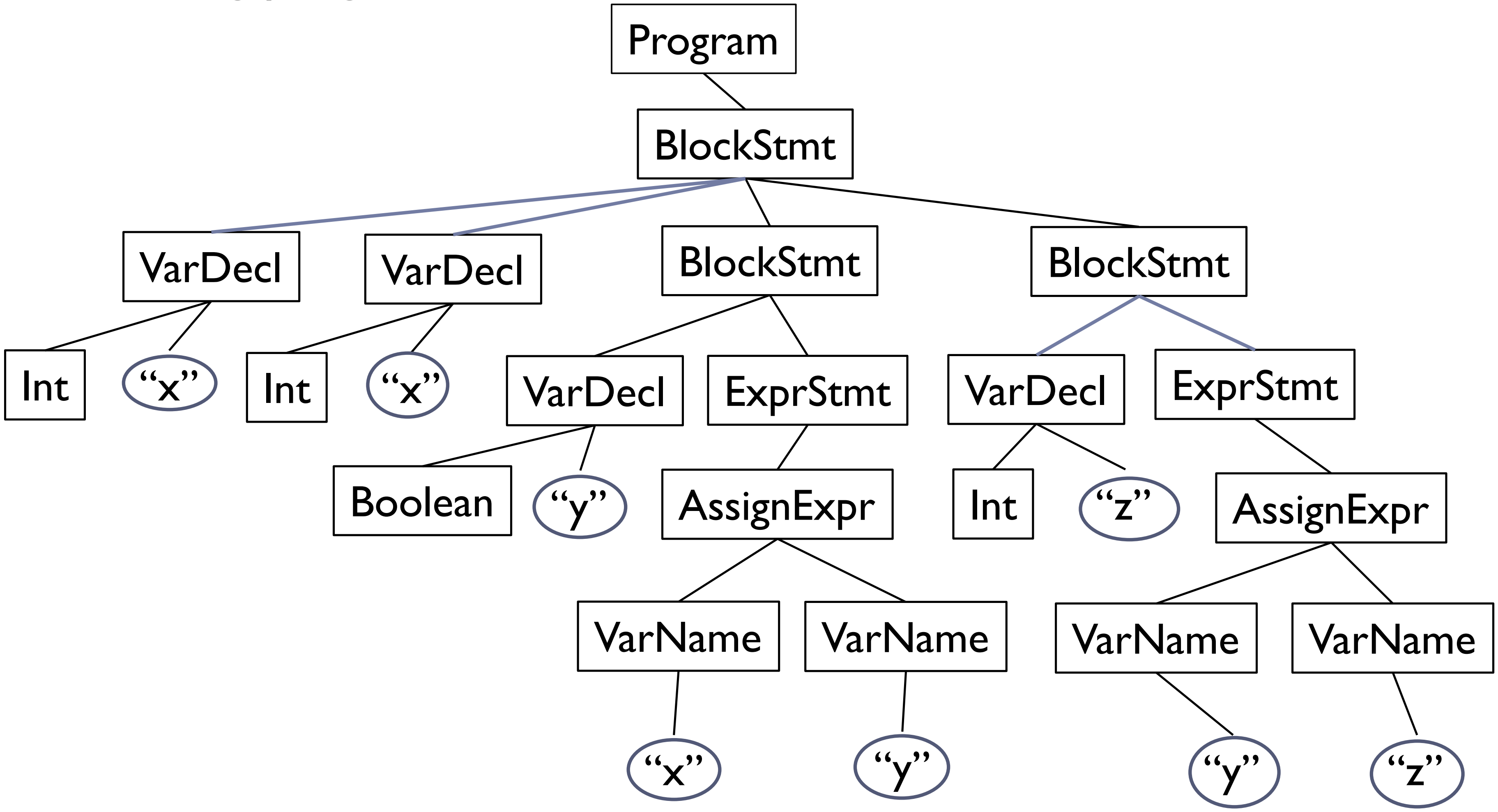
```
// ArrayIndex : LHSExpr ::= Base:Expr  Index:Expr;
public void ArrayIndex.typecheck() {
    // first recursively check the base and the index
    getBase().typecheck();
    getIndex().typecheck();
    // ensure that the base expression's type is an array type
    if  (!(getBase().type() instanceof 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");
}
```

Consider x[0]:
x needs to be an array name

Consider x[j]:
j needs to be an integer

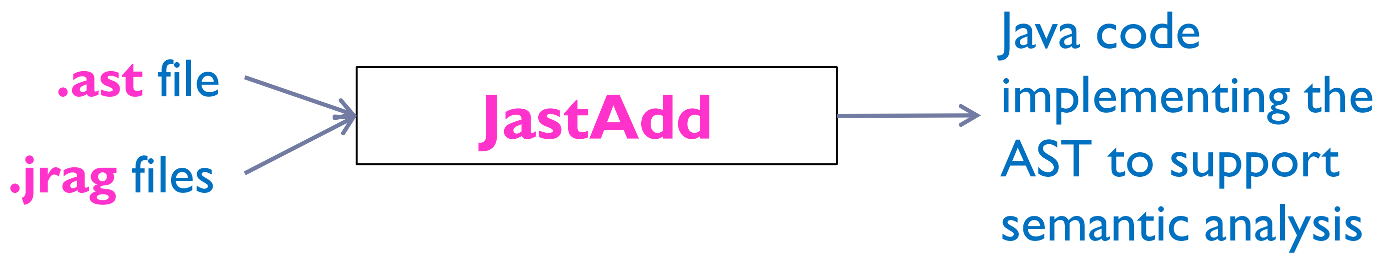
Test Yourself 4.5

What errors will be found by the semantic analyser for the following program?



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 85) and inter-type method declarations (e.g., all the namecheck() and typecheck() methods)
- It is better to group inter-type declarations and attribute grammars that logically belong together in one aspect in one .jrag file.



Finally

- The attribute grammar and the inter-type declarations have defined all the methods needed for semantic analysis.
- The top level code for a semantic analyser:
  - First, call Program.namecheck() to do scope checking.
  - Second, call Program.typecheck() to do type checking.
- If there is no error, we can start code generation.

What else do we learn?
For any repetitive programming work, build a tool to automate!

Test Yourself 4.6

1. What does a semantic analyser 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 26 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. When are type descriptors added to the AST?

Appendix: Scope of a Variable

- Different languages have different scope rules.

- A site for Java:

http://www.startertutorials.com/corejava/scope-lifetime-variables-java.html

- A site for C++:

http://www.learncpp.com/cpp-tutorial/4-1a-local-variables-and-local-scope/

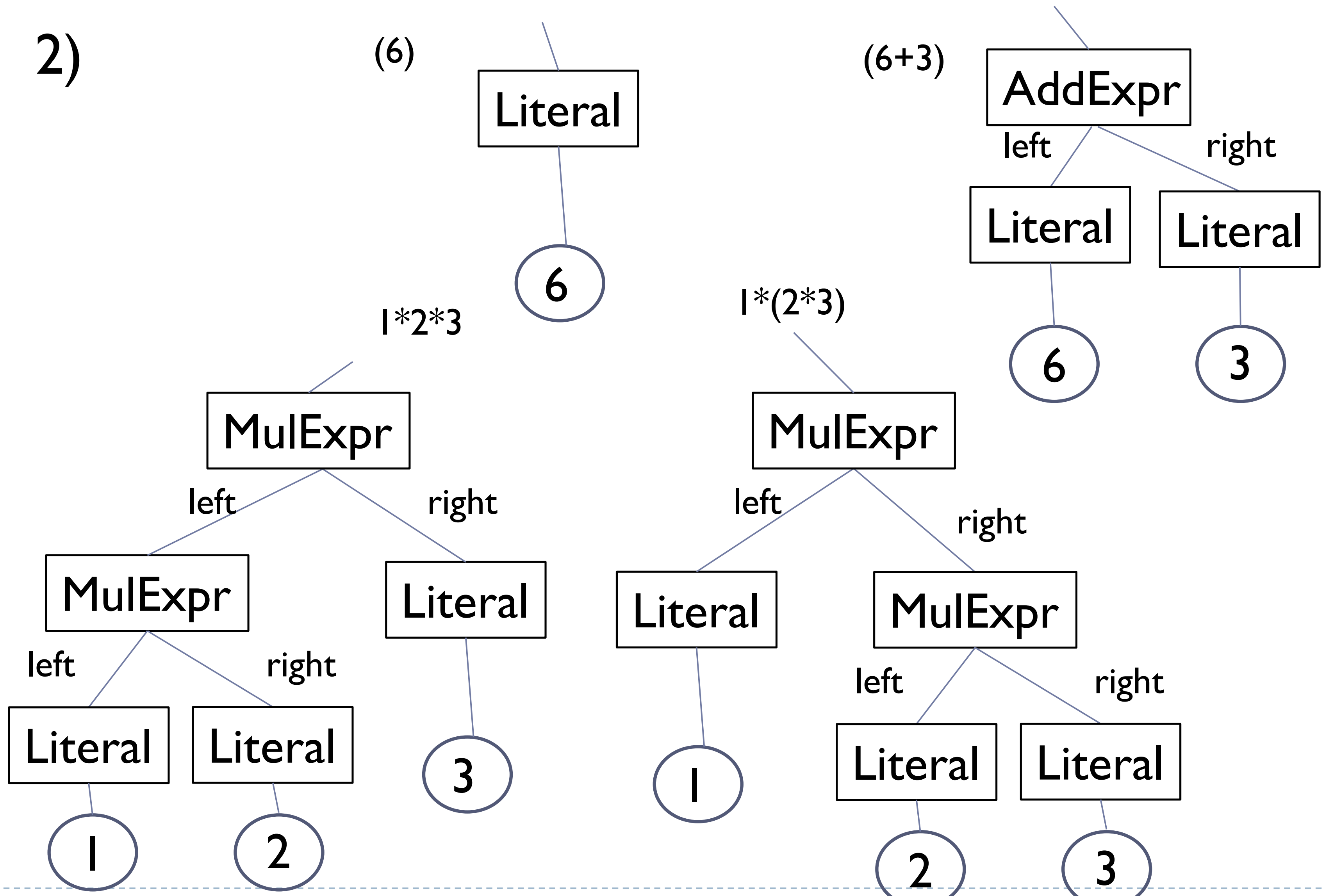
Test Yourself 4.1 (answers)

1)

```
abstract BoolExpr;
abstract BinBExpr : BoolExpr ::= Left : BoolExpr  Right :
   BoolExpr ;
OrExpr : BinBExpr;
AndExpr : BinBExpr;
Literal : BoolExpr ::= <Value:boolean> ;
NotExpr : BoolExpr ::= Operand : BoolExpr;
```

Test Yourself 4.1 (answers)

2)



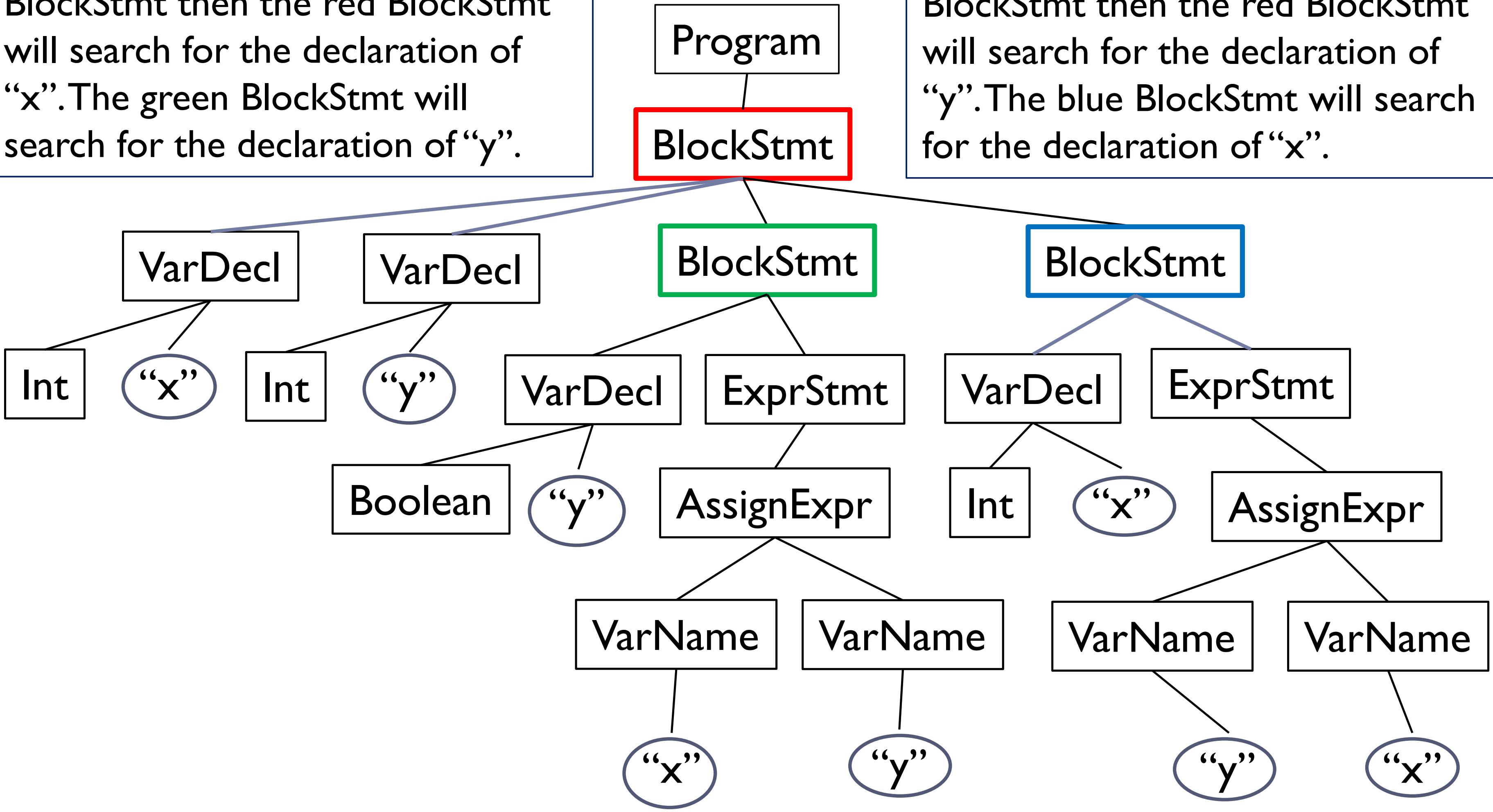
Test Yourself 4.2 (answers)

```
syn  int  Expr.value();
eq  AddExpr.value() = getLeft().value() + getRight().value();
eq  MulExpr.value() = getLeft().value() * getRight().value();
eq  Literal.value() = getValue();
```

Test Yourself 4.3 (answers)

For assignment x = y, first the green BlockStmt then the red BlockStmt will search for the declaration of "x". The green BlockStmt will search for the declaration of "y".

For assignment y = x, first the blue BlockStmt then the red BlockStmt will search for the declaration of "y". The blue BlockStmt will search for the declaration of "x".



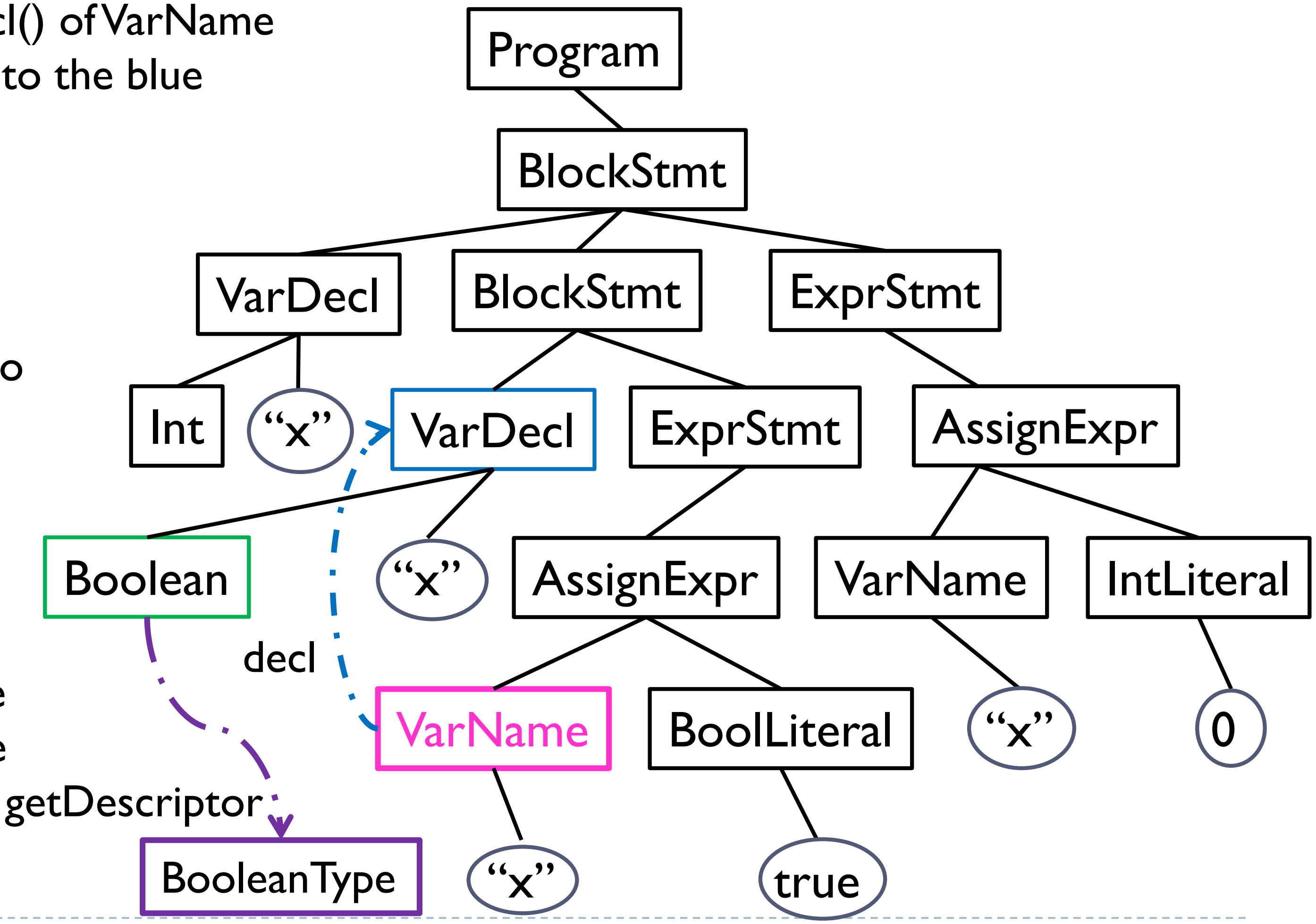
Test Yourself 4.4 (answers)

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

1. Computation of decl() of VarName returns the reference to the blue VarDecl node.

2. Computation of getTypeName() of the blue VarDecl node returns the reference to the green Boolean node.

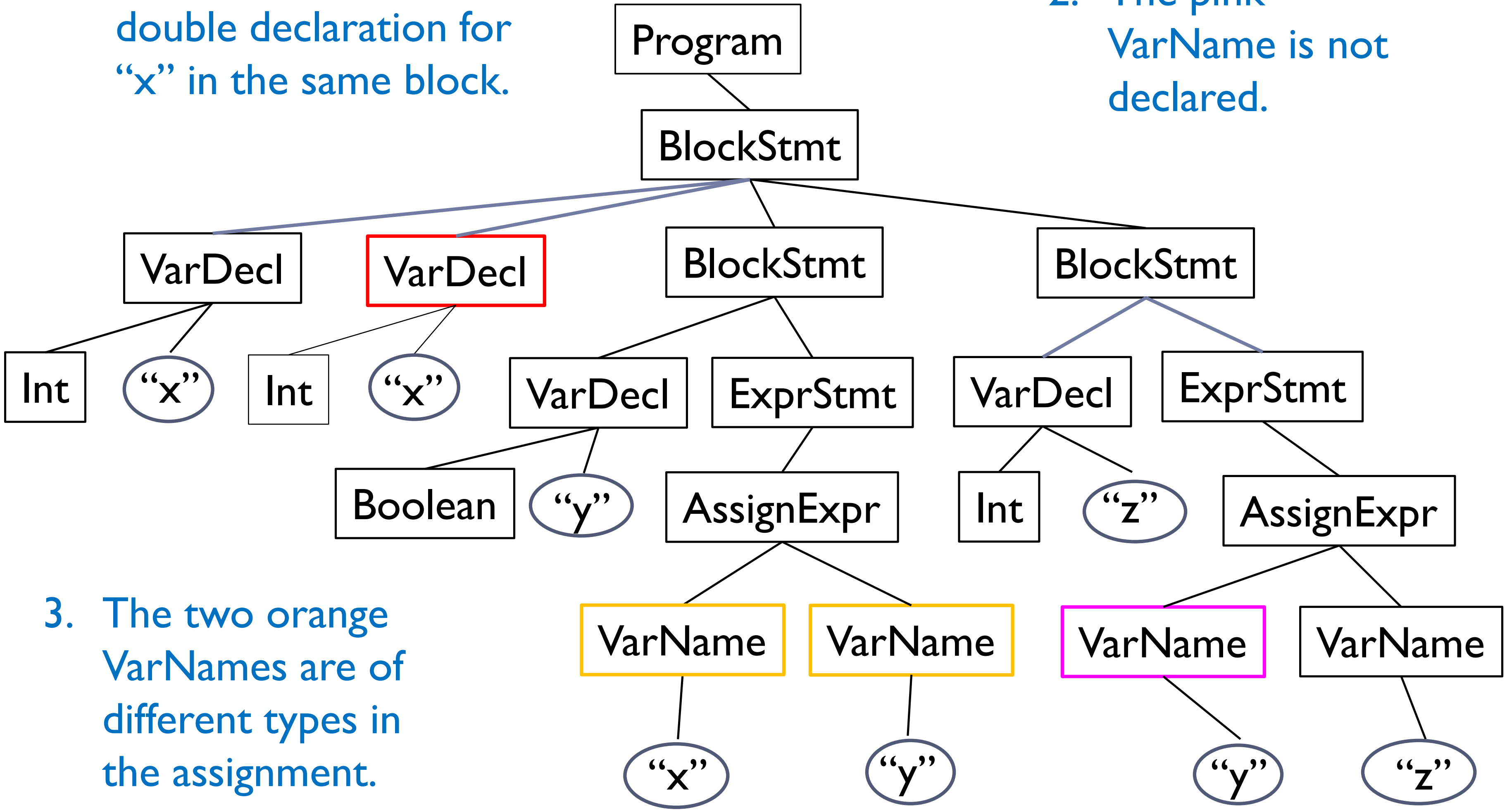
3. Computation of getDescriptor() of the green Boolean node creates and returns the reference to the purple BooleanType node.



Test Yourself 4.5 (answers)

1. The red VarDecl is a double declaration for "x" in the same block.

2. The pink VarName is not declared.



3. The two orange VarNames are of different types in the assignment.