

04-0: **Abstract Syntax Tree (AST)**

- Parse trees tell us exactly how a string was parsed
- Parse trees contain more information than we need
  - We only need the basic shape of the tree, not where every non-terminal is
  - Non-terminals are necessary for parsing, not for meaning
- An Abstract Syntax Tree is a simplified version of a parse tree – basically a parse tree without non-terminals

04-1: **Parse Tree Example**

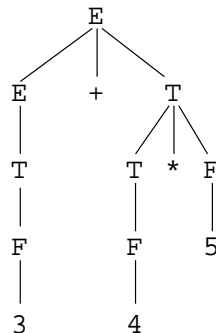
$E \rightarrow E + T$   
 $E \rightarrow T$   
 $T \rightarrow T * F$   
 $T \rightarrow F$   
 $F \rightarrow \text{num}$

Parse tree for  $3 + 4 * 5$

04-2: **Parse Tree Example**

$E \rightarrow E + T$   
 $E \rightarrow T$   
 $T \rightarrow T * F$   
 $T \rightarrow F$   
 $F \rightarrow \text{num}$

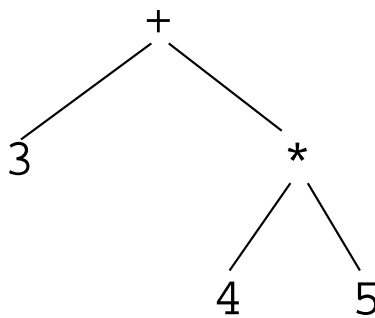
Parse tree for  $3 + 4 * 5$



04-3: **Abstract Syntax Tree Example**

$E \rightarrow E + T$   
 $E \rightarrow T$   
 $T \rightarrow T * F$   
 $T \rightarrow F$   
 $F \rightarrow \text{num}$

Abstract Syntax Tree for  $3 + 4 * 5$



04-4: **AST – Expressions**

- Simple expressions (such as integer literals) are a single node

- Binary expressions (+, \*, /, etc.) are represented as a root (which stores the operation), and a left and right subtree

- $5 + 6 * 7 + 8$  (on whiteboard)

## 04-5: AST – Expressions

- What about parentheses? Do we need to store them?

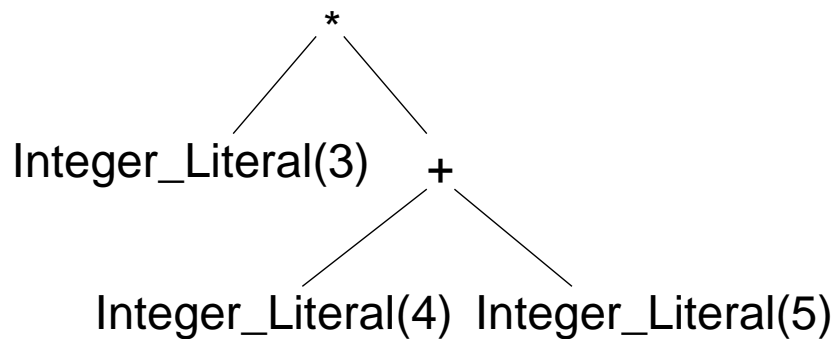
## 04-6: AST – Expressions

- What about parentheses? Do we need to store them?
  - Parenthesis information is store in the shape of the tree
  - No extra information is necessary

$3 * (4 + 5)$

## 04-7: AST – Expressions

$3 * (4 + 5)$

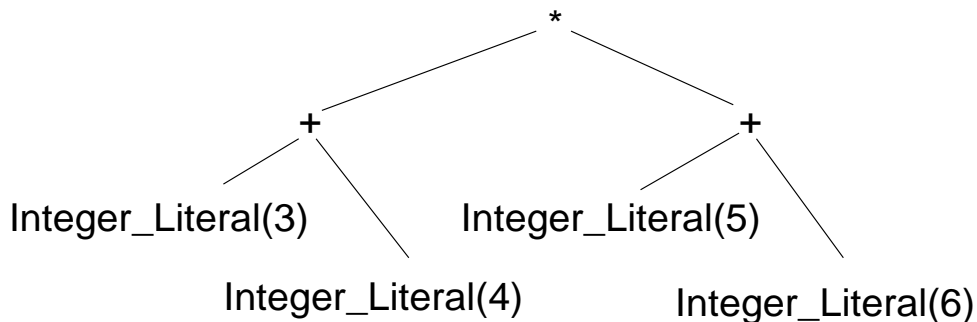


## 04-8: AST – Expressions

$(3 + 4) * (5 + 6)$

## 04-9: AST – Expressions

$(3 + 4) * (5 + 6)$



## 04-10: AST – Expressions

$((4))$

## 04-11: AST – Expressions

$((4))$

Integer\_Literal(4)

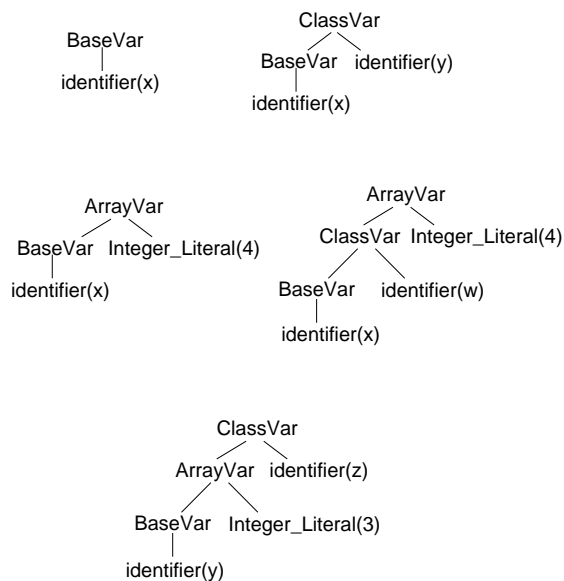
## 04-12: AST – Variables

- Simple variables (which we will call *Base Variables*) can be described by a single identifier.
- Instance variable accesses ( $x.y$ ) require the name of the base variable ( $x$ ), and the name of the instance variable ( $y$ ).
- Array accesses ( $A[3]$ ) require the base variable ( $x$ ) and the array index ( $3$ ).
- Variable accesses need to be extensible
  - $x.y[3].z$

## 04-13: AST – Variables

- **Base Variables** Root is “BaseVar”, single child (name of the variable)
- **Class Instance Variables** Root is “ClassVar”, left subtree is the “base” of the variable, right subtree is the instance variable name
- **Array Variables** Root is “ArrayVar”, left subtree is the “base” of the variable, right subtree is the index

## 04-14: AST – Variables



## 04-15: AST – Instance Variables

```

class simpleClass {
    int a;
    int b;
}
class complexClass {
    int u;
    simpleClass v;
}
void main() {
    complexClass x;
    x = new complexClass();
    x.v = new simpleClass();

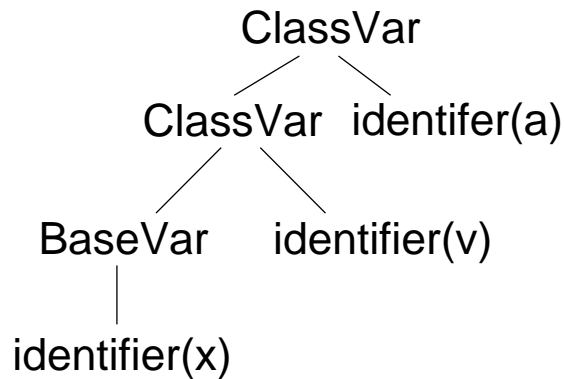
    x.v.a = 3;
}
  
```

## 04-16: AST – Instance Variables

x.v.a

## 04-17: AST – Instance Variables

x.v.a

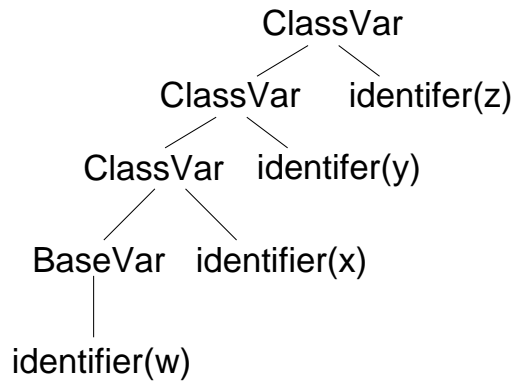


04-18: AST – Instance Variables

w.x.y.z

04-19: AST – Instance Variables

w.x.y.z

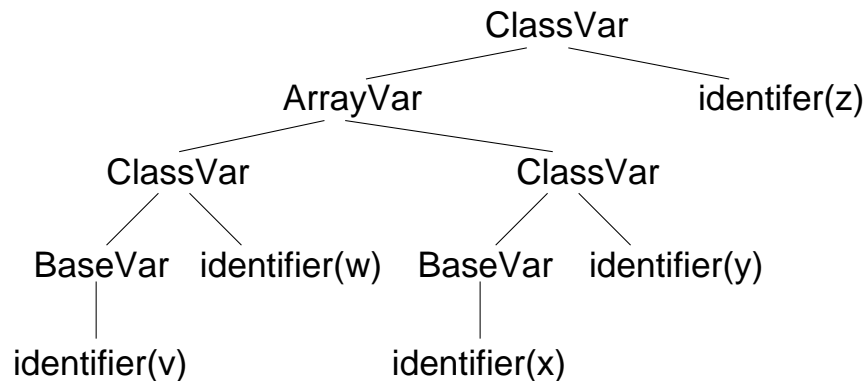


04-20: AST – Instance Variables

v.w[x.y].z

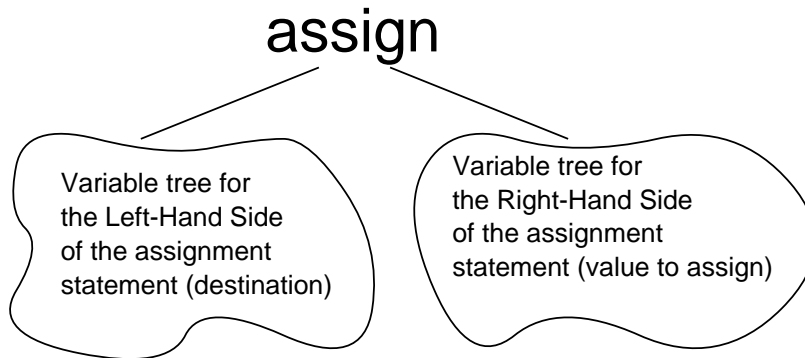
04-21: AST – Instance Variables

v.w[x.y].z



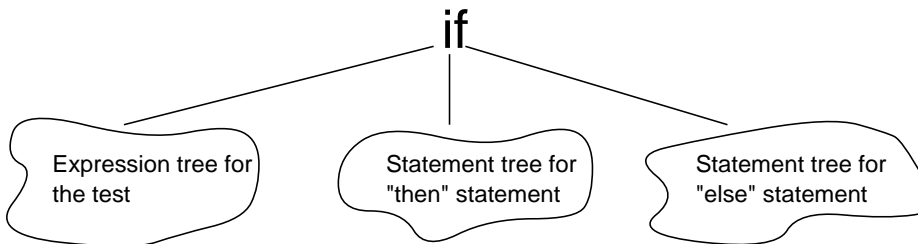
04-22: AST – Statements

- **Assignment Statement** Root is “Assign”, children for left-hand side of assignment statement, and right-hand side of assignment statement



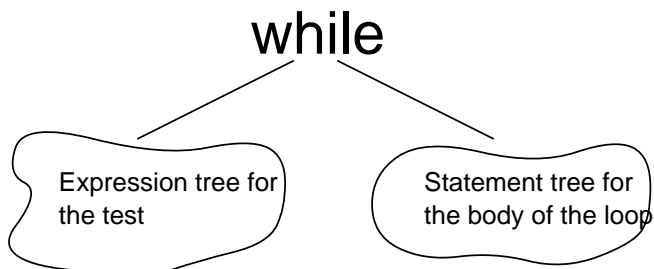
## 04-23: AST – Statements

- **If Statement** Root is “If”, children for test, “then” clause, and “else” clause of the statement. The “else” tree may be empty, if the statement has no else.



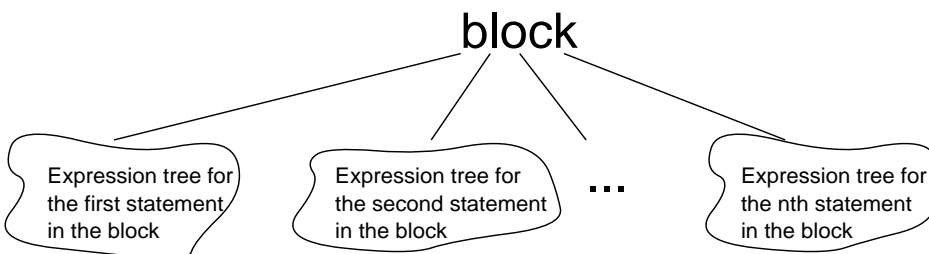
## 04-24: AST – Statements

- **While Statement** Root is “While”, children for test and body of the while loop.



## 04-25: AST – Statements

- **Block Statement** That is, {<statement1>; <statement2>; ... <statementn> }. Block statements have a variable number of children, represented as a Vector of children.



## 04-26: AST – Statements

- **Variable Declarations**

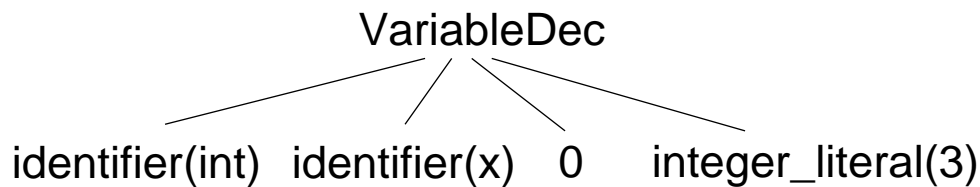
- Non-array variable declaration  
`<TYPE> <NAME>;`
- One-Dimensional array declaration  
`<TYPE> <NAME>[ ];`
- Two-Dimensional array declaration  
`<TYPE> <NAME>[ ][ ];`

## 04-27: AST – Statements

- **Variable Declarations** ASTs for variable declarations have 4 children:
  - An identifier, for the type of the declared variable
  - An identifier, for the name of the declared variable
  - An integer, for the dimensionality of the array (non-array variables have dimension 0)
  - An expression tree, which represents the initial value (if any)

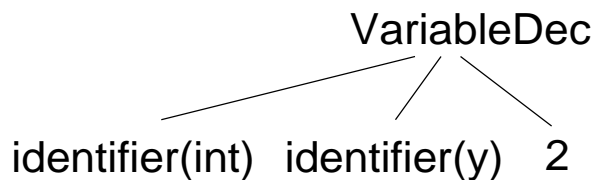
## 04-28: Variable Declarations

```
int x = 3;
```



## 04-29: Variable Declarations

```
int y[ ][ ];
```



## 04-30: New Array Expressions

- New Array expressions are similar to variable expressions:
  - Single dimensional array  
`new int[3];`
  - Two dimensional array

```
new int[3][[]];
```

- Three dimensional array

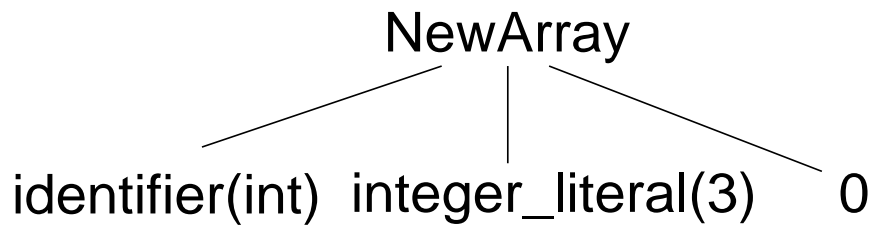
```
new int[4][[][]];
```

#### 04-31: New Array Expressions

- New Array expressions have 3 children
  - Type of array to allocate
  - Number of elements in the new array
  - Dimensionality of each array element

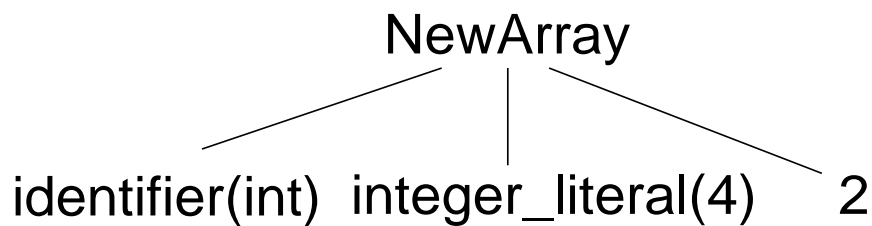
#### 04-32: New Array Expressions

```
new int[3];
```



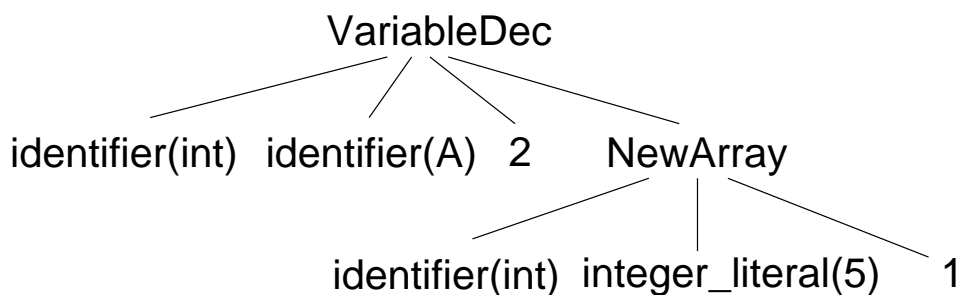
#### 04-33: New Array Expressions

```
new int[4][[][]];
```



#### 04-34: New Array Expressions

```
int A[][] = new int[5][[]];
```



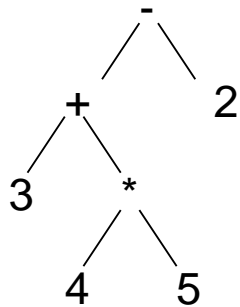
#### 04-35: Representing Trees in Java

- Each “Subtree” is an instance variable
- For trees with variable numbers of children (function calls, etc.), use arrays or Vectors
- Access instance variables using accessor methods

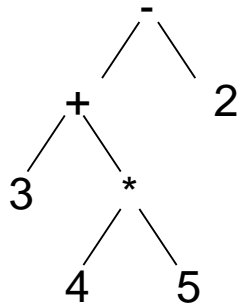
04-36: **Representing Trees in Java**

- Expression Trees
  - Abstract ASTExpression superclass
  - Integer Literal Expression
  - Operator Expression

Go over code for ASTExpression, ASTIntegerLiteralExpression, ASTOperatorExpression 04-37: **Representing Trees in Java**

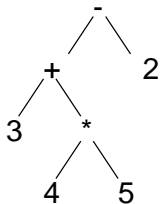


- How could we build this tree?

04-38: **Representing Trees in Java**

```

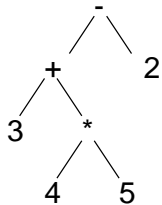
ASTExpression t1, t2, tree;
t1 = new ASTOperatorExpression(ASTIntegerLiteral(4),
                             ASTIntegerLiteral(5), "**");
t2 = new ASTOperatorExpression(ASTIntegerLiteral(3),
                             t1, "+");
tree = new ASTOperatorExpression(t2,
                                ASTIntegerLiteral(2),
                                ASTOperatorExpression.MINUS);
  
```

04-39: **Representing Trees in Java**

- We can extract the integer value 2:

```
int value = ((ASTIntegerLiteral)
            ((ASTOperatorExpression) tree).right()
            ).value();
```

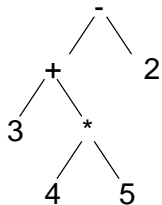
#### 04-40: Representing Trees in Java



- We can extract the integer value 3:

```
int value = ((ASTIntegerLiteral)
            ((ASTOperatorExpression)
             ((ASTOperatorExpression) tree).left()
             ).left()
            ).value();
```

#### 04-41: Representing Trees in Java



- We can extract the integer value 5:

```
int value = ((ASTIntegerLiteral)
            ((ASTOperatorExpression)
             ((ASTOperatorExpression)
              ((ASTOperatorExpression) tree).left()
              ).right()
             ).right()
            ).value();
```

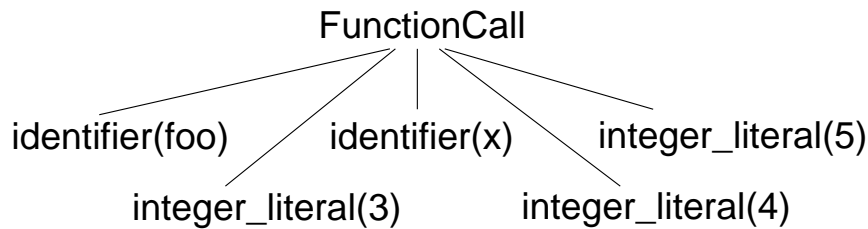
#### 04-42: Representing Trees in Java

- A few extra details ...
  - All AST nodes will contain a “line” instance variable
    - Accessed through the `line()` and `setline()` methods
    - Notes which line on the input file the node appeared on
  - All AST nodes will contain an “accept” method (explained in the next few slides)

#### 04-43: Representing Trees in Java

- Trees with variable numbers of children

```
f00(3, x, 4, 5);
```



## 04-44: Variable # of Children

- Constructor which creates no children
  - Also a constructor that contains a single child
- Add children with `addElement` method
- Find children with `elementAt` method
- Find # of children with `size` method

Put up `ASTFunctionCallStatement.java`

## 04-45: Traversing Trees in Java

- Want to write a function that calculates the value of an expression tree
  - Function that takes as input an expression
  - Returns the value of the expression

## 04-46: Traversing Trees in Java

```

int Calculate(ASTExpression tree) {
    ...
}
  
```

- How do we determine what kind of expression we are traversing
- (Integer Literal, or Operator)?

## 04-47: Traversing Trees in Java

```

int Calculate(ASTExpression tree) {
    if (tree instanceof ASTIntegerLiteral)
        return ((ASTIntegerLiteral)tree).value();
    else {
        int left = Calculate(((ASTOperatorExpression) tree).left());
        int right = Calculate(((ASTOperatorExpression) tree).right());
        switch ((ASTOperatorExpression) tree.operator()) {
            case ASTOperatorExpression.PLUS:
                return left + right;
            case ASTOperatorExpression.MINUS:
                return left - right;
            case ASTOperatorExpression.TIMES:
                return left * right;
            case ASTOperatorExpression.DIVIDE:
                return left / right;
        }
    }
}
  
```

## 04-48: Traversing Trees in Java

- Using “instance of”, and all of the typecasting, is not very elegant

- There is a better way – Visitor Design Pattern

#### 04-49: Traversing Trees in Java

```
int Calculate(ASTExpression tree) {
    if (tree instanceof ASTIntegerLiteral)
        return CalculateIntegerLiteral((ASTIntegerLiteral)tree);
    else if (tree instanceof ASTOperatorExpression)
        return CalculateOperatorExpression((ASTOperatorExpression) tree);
    else
        return -1; /* error! */
}
```

#### 04-50: Traversing Trees in Java

```
int CalculateOperatorExpression(ASTOperatorExpression tree) {
    int left = Calculate(tree.left());
    int right = Calculate(tree.right());
    switch (tree.operator()) {
        case ASTOperatorExpression.PLUS:
            return left + right;
        case ASTOperatorExpression.MINUS:
            return left - right;
        case ASTOperatorExpression.TIMES:
            return left * right;
        case ASTOperatorExpression.DIVIDE:
            return left / right;
    }
}

int CalculateIntegerLiteral(ASTIntegerLiteral tree) {
    return tree.value();
}
```

#### 04-51: Virtual Function Review

- Quick Review of virtual functions
- See files `Shape.java`, `Circle.java`, `Square.java` on other screen

```
Shape Shapes[];
...
for (i=0; i<Shapes.size; i++)
    Shapes[i].draw();
```

#### 04-52: Traversing Trees in Java

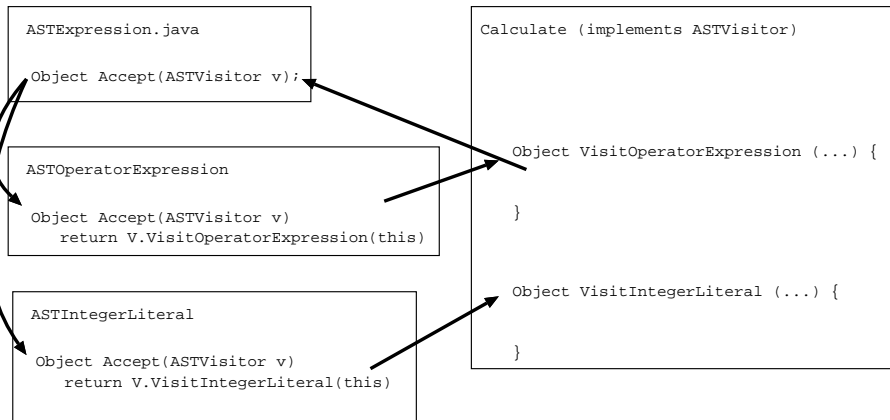
- Using “instance of”, and all of the typecasting, is not very elegant
- There is a better way – Visitor Design Pattern
  - A Visitor is used to traverse the tree
  - Visitor contains a Visit method for each kind of node in the tree
  - The visit method determines how to process that node
  - Each node in the AST has an “accept” method, which calls the appropriate visitor method, passing in a pointer to itself

#### 04-53: Traversing Trees in Java

- Each node in the AST contains an “accept” method
  - Takes as input a visitor
  - Calls the appropriate method of the visitor to handle the node, passing in a pointer to itself
  - Returns whatever the visitor tells it to return

Put up examples of Expression AST w/ "accept" methods

#### 04-54: Traversing Trees in Java



#### 04-55: Visitor Interface

- Visitor Interface for Expression Trees

```

public interface ASTVisitor {
    public Object VisitIntegerLiteral(ASTIntegerLiteral literal);
    public Object VisitOperatorExpression(ASTOperatorExpression opexpr);
}
  
```

#### 04-56: Visitor Implementation

- Write a Visitor to calculate the value of an expression tree
  - Implement VisitIntegerLiteral and VisitOperatorExpression methods

```

public Object VisitIntegerLiteral(ASTIntegerLiteral literal) {
    ...
}
  
```

#### 04-57: Visitor Implementation

- Write a Visitor to calculate the value of an expression tree
  - Implement VisitIntegerLiteral and VisitOperatorExpression methods

```

public Object VisitIntegerLiteral(ASTIntegerLiteral literal) {
    return new Integer(literal.value());
}
  
```

#### 04-58: Visitor Implementation

```

public Object VisitOperatorExpression(ASTOperatorExpression opexpr) {
    ...
}
  
```

#### 04-59: Visitor Implementation

```

public Object VisitOperatorExpression(ASTOperatorExpression opexpr) {
    Object left = opexpr.left().Accept(this);
    Object right = opexpr.right().Accept(this);

    int leftValue = ((Integer) left).intValue();
    int rightValue = ((Integer) right).intValue();
    switch (opexpr.operator()) {
    case ASTOperatorExpression.PLUS:
        return new Integer(leftValue + rightValue);
    }
}
  
```

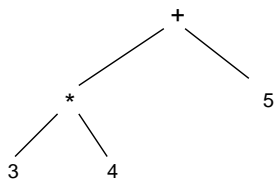
```

case ASTOperatorExpression.MINUS:
    return new Integer(leftValue - rightValue);
case ASTOperatorExpression.MULTIPLY:
    return new Integer(leftValue * rightValue);
case ASTOperatorExpression.DIVIDE:
    return new Integer(leftValue / rightValue);
default:
    System.out.println("ERROR -- Illegal Operator");
    return new Integer(-1);
}
}

```

#### 04-60: More Visitors – Tree Printing

- We'd like to print out expression trees
- Show the structure of the tree itself



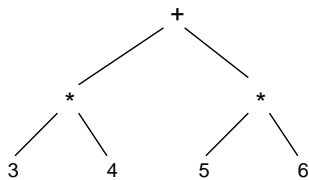
```

+
 *
  3
  4
  5

```

#### 04-61: More Visitors – Tree Printing

- We'd like to print out expression trees
- Show the structure of the tree itself



```

+
 *
  3
  4
 *
  5
  6

```

#### 04-62: Tree Printing

- Maintain a “current indentation level”
- To Print out a Integer Literal
  - Print the value at the current indentation level

- To Print out an operator
  - Print the root of the tree at the current indentation level
  - Print the children at a larger indentation level

Code for Tree Printing on other screen 04-63: **Visitor Overview**

- Each node in the AST has an “accept” method, that takes a visitor as an input parameter, and calls the appropriate method of that visitor. The “accept” method then returns whatever the visit method returns.
- The Visitor has a visit method for each AST node, which handles visiting that node. *Typically*, visit methods call the “accept” method on the subtrees of the node, collecting data from the subtrees. This data is then combined, and returned.
- Visitors often contain instance variables that allow data to be shared among visit methods (such as the “current indentation level” for printing trees)

#### 04-64: **JavaCC Actions**

- Each JavaCC rule is converted into a parsing method
  - Just like a recursive descent parser created by hand
- We can add arbitrary Java code to these methods
- We can also add instance variables and helper methods that every parser method can access

#### 04-65: **JavaCC Actions**

- Adding instance variables

```
PARSER_BEGIN(parserName)
```

```
public class parserName {
```

```
    /* instance variables and helper methods */
```

```
    /* Optional ``main`` method (the ``main``
       method can also be in a separate file) */
```

```
}
```

```
PARSER_END(parserName)
```

#### 04-66: **JavaCC Rules**

```
<return type> <rule name>() :
{
  /* local variables */
}
{
  Rule
  | Rule2
  | ...
}
```

- Each rule can contain arbitrary Java code between { and }

Put up code for parens1.jj file

#### 04-67: JavaCC Rules

- JavaCC rules can also return values
  - Works just like any other method
- Use “;variable; =” syntax to obtain values of method calls

Put up code for parens2.jj

#### 04-68: JavaCC Rules

- Building A JavaCC Calculator
- How would we change the following .jj file so that it computed the value of the expression, as well as parsing the expression?

Put up code for calc.noact.jj

#### 04-69: JavaCC Rules

```
int complete_expression():
{ int result; }
{
    result = expression() <EOL>
        { return result; }
}
```

#### 04-70: JavaCC Rules

```
int factor():
{int value; Token t;}
{
    t = <INTEGER_LITERAL>
        { return Integer.parseInt(t.image); }
    | <MINUS> value = factor()
        { return 0 - value; }
    | <LPAREN> value = expression() <RPAREN>
        { return value; }
}
```

#### 04-71: JavaCC Rules

```
int term():
{Token t; int result; int rhs;}
{
    result = factor() ( (t = <MULTIPLY> | t = <DIVIDE>) rhs = factor()
        { if (t.kind == MULTIPLY)
            result = result * rhs;
          else
            result = result / rhs;
        } )*
    { return result; }
}
```

Swap other screen to calc.jj 04-72: Parsing a term()

- Function to parse a factor is called, result is stored in result

- The next token is observed, to see if it is a \* or /.
- If so, function to parse a factor is called again, storing the result in `rhs`. The value of result is updated
- The next token is observed to see if it is a \* or /.
- ...

#### 04-73: Expression Examples

- 4
- 3 + 4
- 1 + 2 \* 3 + 4

#### 04-74: Input Parameters

- JavaCC rules == function calls in generated parser
- JavaCC rules can have *input parameters* as well as return values
- Syntax for rules with parameters is the same as standard method calls

#### 04-75: Input Parameters

```
void expression():
{
  term() expressionprime()
}

void expressionprime():
{
  <PLUS> term() expressionprime()
  | <MINUS> term() expressionprime()
  { }
```

- What should `<PLUS> term() expressionprime()` return?

#### 04-76: Input Parameters

- What should `<PLUS> term() expressionprime()` return?
  - Get the value of the previous term
  - Add that value to `term()`
  - Combine the result with whatever `expressionprime()` returns
- How can we get the value of the previous term?
- How can we combine the result with whatever `expressionprime()` returns?

#### 04-77: Input Parameters

- What should `<PLUS> term() expressionprime()` return?
  - Get the value of the previous term
  - Add that value to `term()`
  - Combine the result with whatever `expressionprime()` returns

- How can we get the value of the previous term?
  - Have it passed in as a parameter
- How can we combine the result with whatever `expressionprime()` returns?
  - Pass the result into `expressionprime()`, and have `expressionprime()` do the combination

#### 04-78: Input Parameters

```
int expression():
{int firstterm; int result;}
{
    firstterm = term() result = expressionprime(firstterm)
    { return result; }
}

int expressionprime(int firstterm):
{ int nextterm; int result; }
{
    <PLUS> nextterm = term() result = expressionprime(firstterm + nextterm)
    { return result; }
| <MINUS> nextterm = term() result = expressionprime(firstterm - nextterm)
  { return result; }
| { return firstterm; }
}
```

#### 04-79: Building ASTs with JavaCC

- Instead of returning values, return trees
- Call constructors to build subtrees
- Combine subtrees into larger trees

Put up code for calc.noact.jj

#### 04-80: Building ASTs with JavaCC

```
ASTExpression factor():
{ASTExpression value; Token t;}
{
    ...
| t = <INTEGER_LITERAL>
  { return new ASTIntegerLiteral(Integer.parseInt(t.image)); }
    ...
}
```

#### 04-81: Building ASTs with JavaCC

```
ASTExpression factor():
{ASTExpression value; Token t;}
{
    <MINUS> value = factor()
    { return new ASTOperatorExpression(new ASTIntegerLiteral(0),
    value,
    ASTOperatorExpression.MINUS); }
    ...
}
```

#### 04-82: Building ASTs with JavaCC

```
ASTExpression term():
{Token t; ASTExpression result; ASTExpression rhs;}
{
    result = factor() ( (t = <MULTIPLY> | t = <DIVIDE>) rhs = factor()
    { result = new ASTOperatorExpression(result, rhs, t.image);
    } ) *
    { return result; }
}
```

#### 04-83: Project

- Files `ASTxxx.java`
- Tree Printing Visitor

- Driver program

04-84: **Project Hints**

- The Abstract Syntax Tree (`ASTxxx.java`) is pretty complicated. Take some time to understand the structure of sjava ASTs
- There is nothing in the AST for “++” or unary minus, but:
  - `<var>++` is the same as `<var> = <var> + 1`
  - `-<exp>` is the same as `0 - <exp>`