CS453
Abstract Syntax tree (AST)
Visitor patterns

Plan for Today

Abstract Syntax Tree

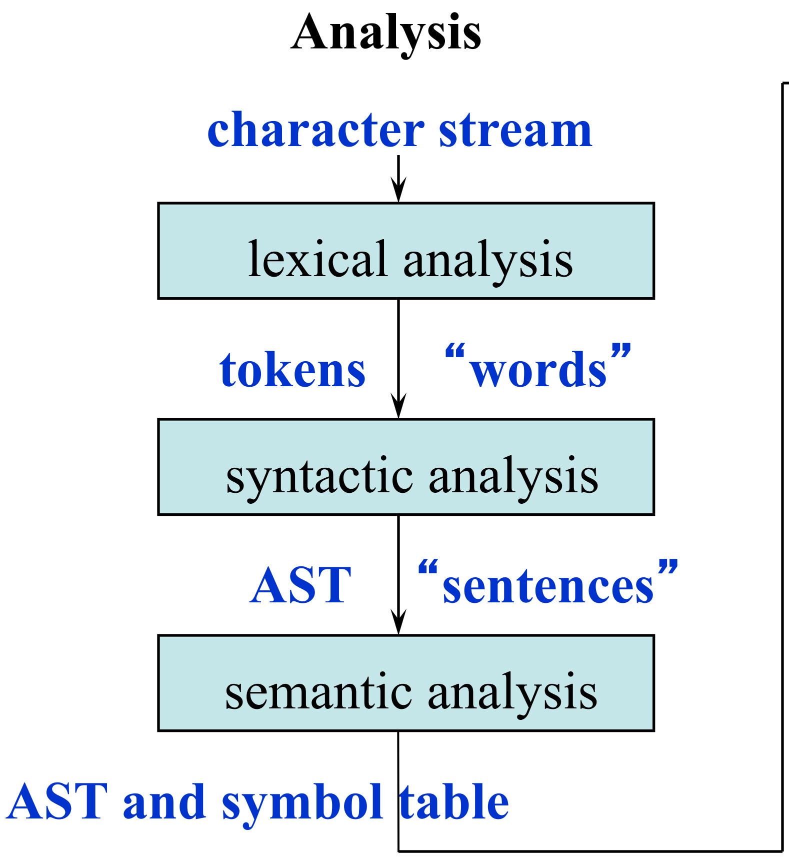
- Example and main idea
- construction with a bottom up parser
- AST for Meggy Java

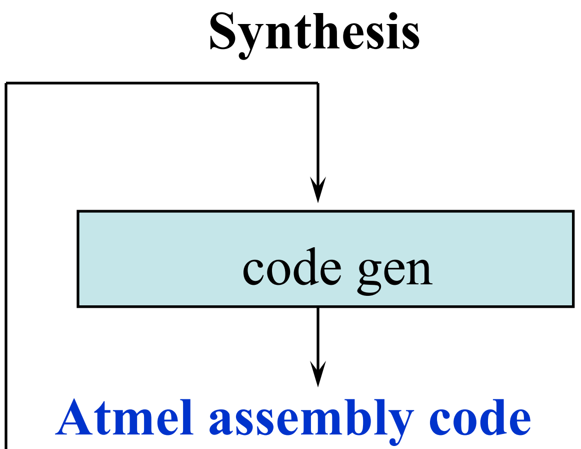
Visitor Design Pattern

- main idea and example
- example reprise using visitor that does traversal
- FAQ about visitors
- Dot visitor
- Other examples including integer and byte expression evaluation

Debugging Ideas

Structure of the MeggyJava Compiler



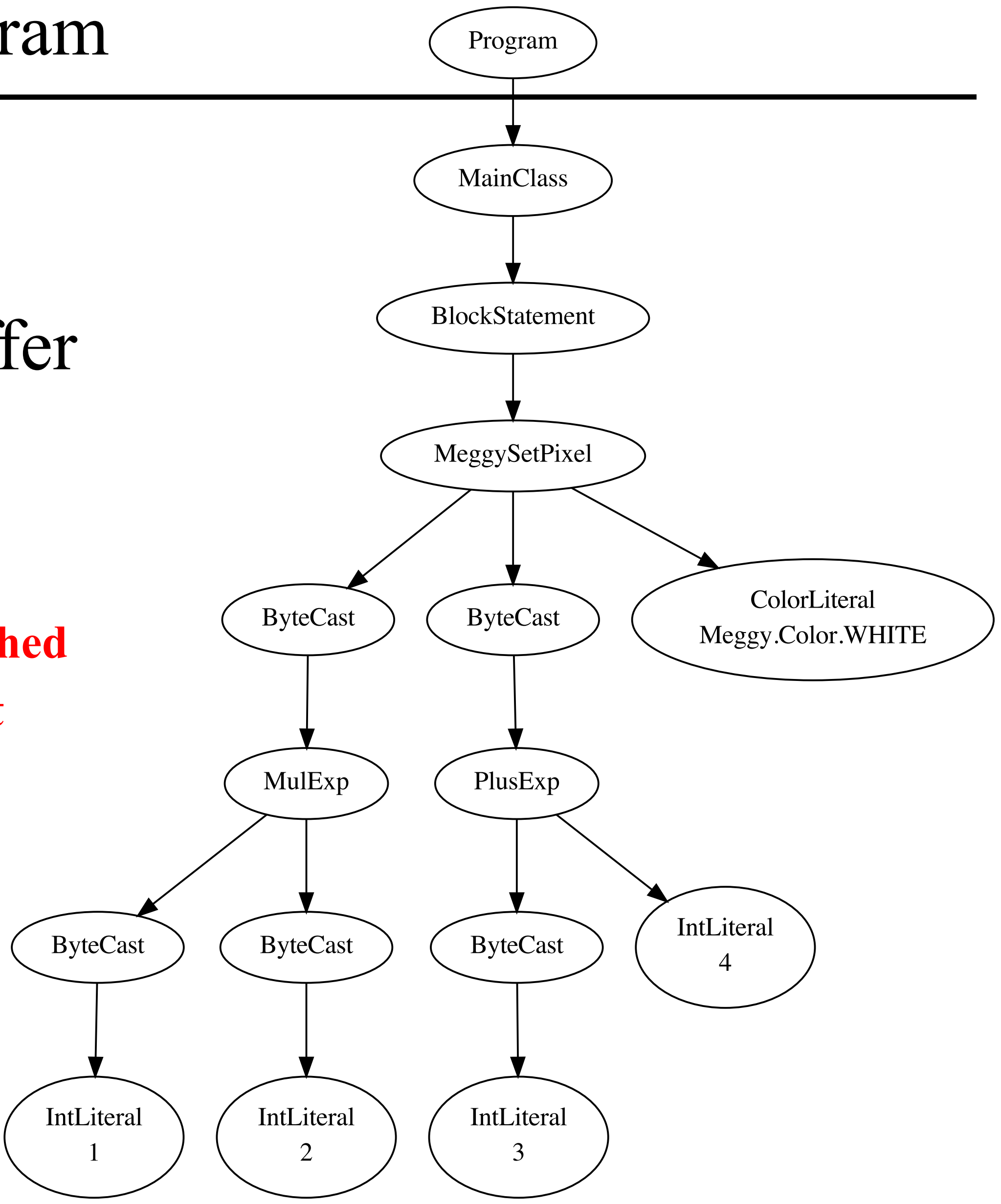


PA1: Write test cases in C++ and MeggyJava, and Atmel warmup
PA2: MeggyJava scanner and setPixel
PA3: add exps and control flow (AST)
PA4: add methods (symbol table)
PA5: add variables and objects

Example program

```
class Byte {
  public static void main(String[] whatever){
          Meggy.setPixel
              (   // Byte multiplication: Byte x Byte -> Int
                  (byte)( (byte)1*(byte)2 ),
                  // Mixed type expression: Byte x Int -> Int
                  (byte)( (byte)3 + 4 ),
                  Meggy.Color.WHITE
              );
  }
}
```

AST of Example Program



How does the AST differ from the parse tree?

Parentheses have been removed
their role -to shape the AST is finished
Some terminals have been pulled out
which?
Some have been pulled up
which?

Grammar Subset and AST Node Hierarchy

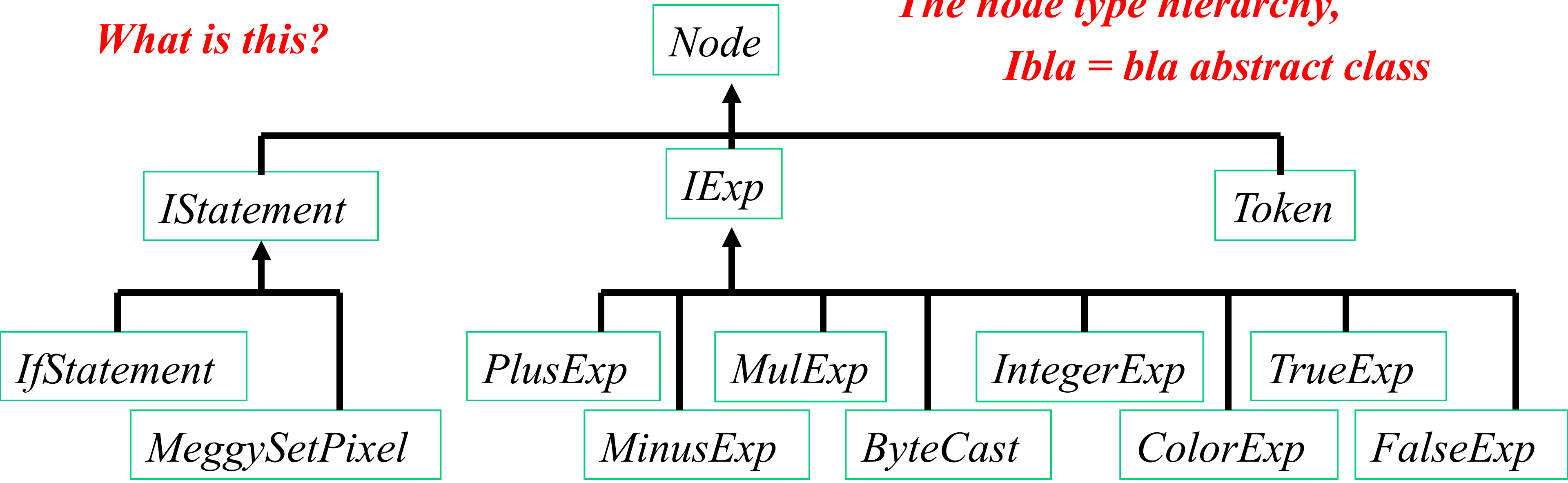
```
Statement ::= "if" "(" Expression ")" Statement "else" Statement
    | "Meggy.setPixel" "(" Expression "," Expression "," Expression ")"

Expression ::=
     Expression ("+" | "-" | "*" ) Expression
    | "(" "byte" ")" Expression
    | <INTEGER_LITERAL> | <COLOR_LITERAL> | "true" | "false"
```

What is this?

The node type hierarchy,

Ibla = bla abstract class



Syntax-directed Construction of AST

The scanner provides line and position of each Symbol in SymbolValue
So the parser can put these in the appropriate nodes of the AST:

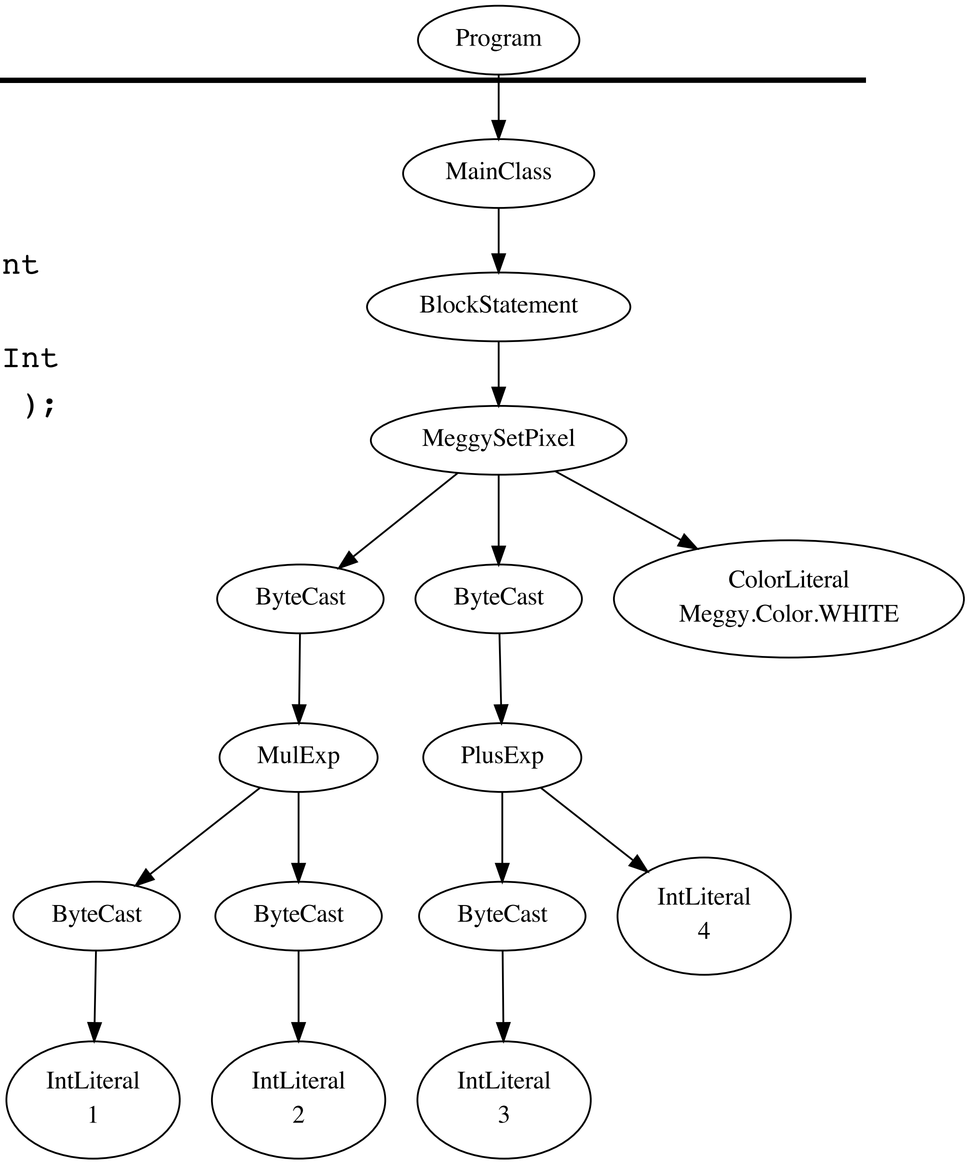
```
Expression ::=
 …
 | exp:a PLUS:op exp:b
   {: RESULT = new PlusExp(a, b, op.line, op.pos); :}

 statement_list ::=
    statement_list:list statement:s
    {: if (s!=null) { list.add(s); }
     RESULT = list; :}

  | /* epsilon */
    {: RESULT = new LinkedList<IStatement>(); :}
  ;
```

Building AST Bottom Up

```
class Byte {
  public static void main(String[] whatever){
    Meggy.setPixel(
      // Byte multiplication: Byte x Byte -> Int
      (byte)( (byte)1*(byte)2 ),
      // Mixed type expression: Byte x Int -> Int
      (byte)( (byte)3 + 4 ), Meggy.Color.WHITE );
  }
}
```



Visitor Design Pattern

Situation

- Want to perform some processing on all items in a data structure, e.g type check or code generate
- Will be adding many different ways to process items depending on the type (class)
- Will not be changing the classes of the data structure itself (much, or at all)

Possibilities

- OO: For each functionality and each class, add a method
  - con: each new functionality is spread over multiple files
  - con: sometimes can't add methods to existing class hierarchy
- Procedural: Use switch statement in one method traversing the data structure
  - pro: keeps all the code for the feature in one place
  - con: can be costly and involve lots of casting
- Visitor design pattern (best of all)

AST and visitors

We will generate an AST instead of directly generating code.

- Why is that a good idea? What can we now do better?

We can walk over this AST multiple times and perform different functions, e.g. Create symbol table, Check types, Generate code

We will then traverse the AST for each particular need using visitors

each node of the AST has an accept method, that calls an appropriate visitor method, e.g. plusExp.accept() calls visitPlusExp()

Class hierarchy is USEFUL, because we only override a few methods: the ones that differ from standard behavior

Visit, In , Out

When visiting the AST, we encounter a node for the first time (In encounter) and we encounter the node for the last time (Out encounter). These encounters are often associated with certain actions:

```
Visitor::visitXYZ(node) {
        inXYZ(node);
        for each child c of node in left to right order
           c.accept(this);
        outXYZ(node);
   }
```

inXYZ is called when the node is first encountered in the DFLR walk, and outXYZ is called when the node is left behind in the DFLR walk.

This is often sufficient for code generation purposes (+,-,*,setPixel), but not always: (if, while, &&). WHY NOT?

Example Use of the visitor design pattern

```
// in driver:
ast_root.accept(new AVRgenVisitor(outfilehandle));

// in AST class MulExp
public void accept(Visitor v)  { v.visitMulExp(this); }

// in class DepthFirstVisitor
public void inMulExp(MulExp node)  { defaultIn(node); }
public void outMulExp(MulExp node) { defaultOut(node); }
public void visitMulExp(MulExp node){
  inMulExp(node);
  if(node.getLExp() != null) node.getLExp().accept(this);
  if(node.getRExp() != null) node.getRExp().accept(this);
  outMulExp(node);
}

// in code generator ← This is YOUR job
public void outMulExp(MulExp node) { // overrides default
   // gen code to pop operands, do the *, push the result
}
```

FAQ, Debugging Ideas

Check out your recitation PA0 Part3 example. It tells you a lot!!

How do I associate data with a node in the AST if I can't add fields to the node classes?

What if I want to do the same thing on each node?
What if I only need to do something on certain nodes?

Debugging

System.out.println in parser actions

Break points in visitor methods

Code Structure

In driver, first call the parser to get an AST:

```
mj_ast_parser parser = new mj_ast_parser(lexer);
ast.node.Node ast_root = (ast.node.Node)parser.parse().value;
```

Next create a dot file for the AST for debugging purposes:

```
java.io.PrintStream astout =  new java.io.PrintStream(…);
ast_root.accept(new DotVisitor(new PrintWriter(astout)));
```

Finally, create Type-Checker and an AVRgenVisitor instances:

```
java.io.PrintStream avrsout = new java.io.PrintStream(…);
ast_root.accept(new AVRgenVisitor(new PrintWriter(avrsout)));
System.out.println("Printing Atmel assembly to " + filename + ".s");
```