Compiler Compiler Tutorial

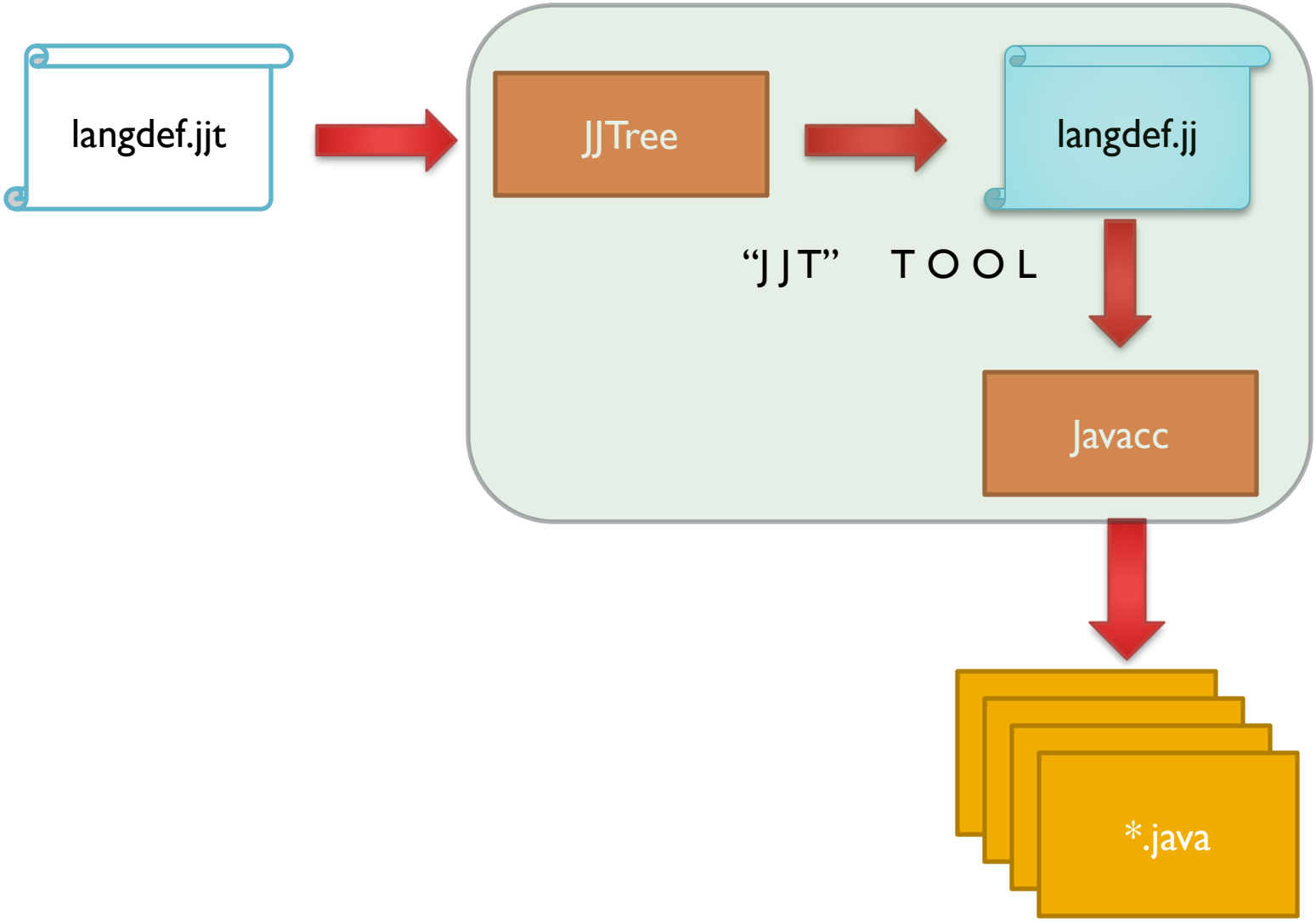
CSA2010 – Compiler Techniques

Gordon Mangion

Topics

- Quick revision
  - Compiler modules
  - Javacc
  - Worksheet
- Visitor Pattern
- Semantic Analysis
- Code generation
- The assignment (VSL)

JJTree



Worksheet

Digit ::= [“**0**” - “**9**”]

Number ::= Digit+

Plus ::= “+”

Minus ::= “**-**”

Op ::= Plus | Minus

Expression ::= Number { Op Number }

Worksheet

```
TOKEN:
{
< NUMBER: <DIGIT>+ >
| < Digit: [“0” - “9”] >
| < PLUS:“+” >
| < MINUS:“-” >
}
```

Digit ::= [“**0**” - “**9**”]
Number ::= Digit+
Plus ::= “+”
Minus ::= “-”

Worksheet

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

void Expression(): {}
{
   < Number >
   (Op() < Number >)*
}
```

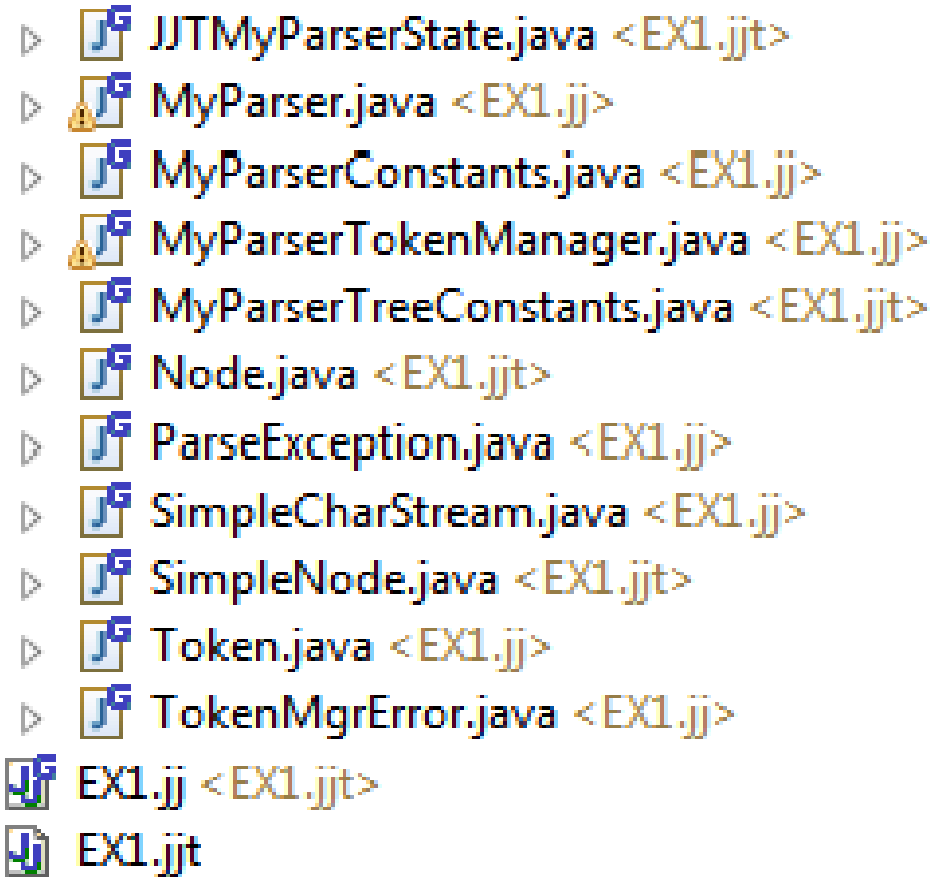
Op ::= Plus | Minus

Expression ::=
Number { Op Number }

Worksheet

```
PARSER_BEGIN(MyParser)
...
MyParser parser = new MyParser(System.in);
try {
   parser.Expression();
   System.out.println(“Parsed!");
} catch (Exception e) {
      System.out.println("Oops!");
      System.out.println(e.getMessage());
}
...
PARSER_END(MyParser)
```

Generated Sources



Worksheet – Evaluating

```
Token Op() :{ Token t; }
{
   (t = < PLUS > | t = < MINUS >)
   { return t; }
}
```

Worksheet

```
int Expression(): { Token t, op; int n;}
{
    t = < NUMBER >
    {
        n = Integer.parseInt( t.image );
    }
    ( op=Op()
    t = < NUMBER > {
    if(op.image.equals("+"))
            n += Integer.parseInt( t.image );
    else
            n -= Integer.parseInt( t.image ); }
    )*
    { return n; }
}
```

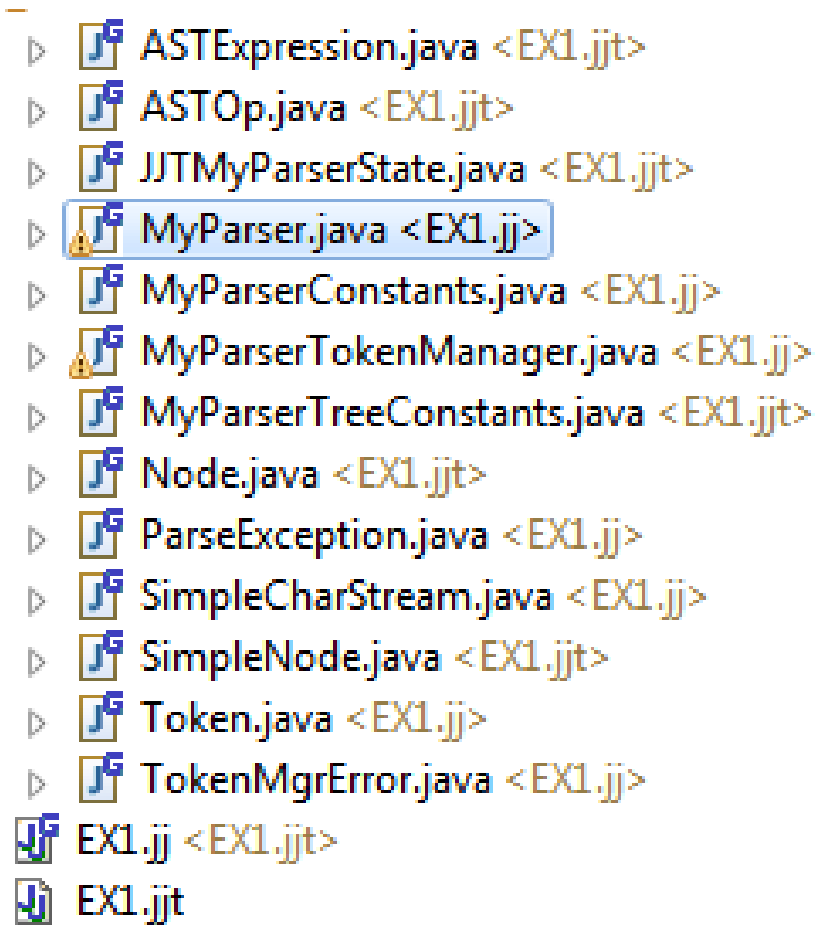
Worksheet

```
PARSER_BEGIN(MyParser)

public class MyParser
{
...
    MyParser parser = new MyParser(System.in);
    try
    {
      int n = parser.Expression();
...

PARSER_END(MyParser)
```

Building the AST

```
options
{
   STATIC=false;
   MULTI=true;
   BUILD_NODE_FILES=true;
   NODE_USES_PARSER=false;
   NODE_PREFIX="AST";
   …
}
```

Generated Code



Worksheet

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

SimpleNode Expression(): {}
{
   < Number >
   (Op() < Number >)*
   { return jjtThis; }
}
```

Op ::= Plus | Minus

Expression ::=
Number { Op Number }

Worksheet

```
PARSER_BEGIN(MyParser)

public class MyParser
{
...
    MyParser parser = new MyParser(System.in);
    try
    {
      SimpleNode rootNode = parser.Expression();
...

PARSER_END(MyParser)
```

Example

Digit ::= [“**0**” - “**9**”]

Number ::= Digit+

Factor ::= Expression | Number

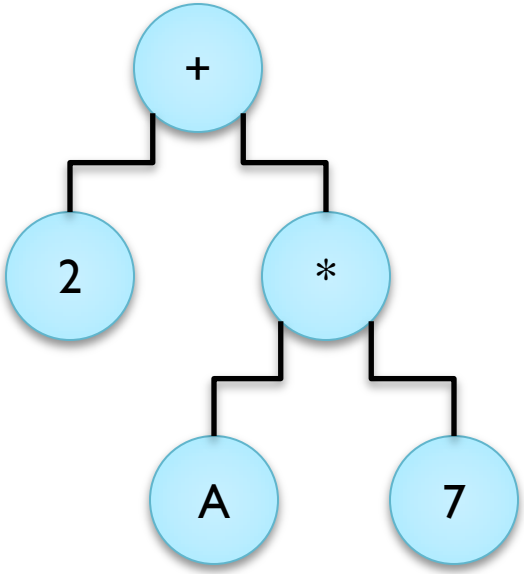
Term ::= Factor [“*” | “/” Factor]

Expression ::= Term { “**+**” | “**-**” Term }

Start ::= Expression

The Visitor Design Pattern

- The Problem



  - Number of operations to be performed on each node
  - Options
    - Implement each operation inside each node
    - Make use of visitor pattern

The Visitor Design Pattern

- Consider one Node
  - Printing Operation

- We can implement the operation in a separate class e.g. PrintVisitor

```
PrintVisitor

 void PrettyPrint( Plus )
{
   …
}
```

- This can be done for all type of nodes

The Visitor Design Pattern

- Modification on node

```
Plus

void Print( Visitor  p_visitor)
{
    p_visitor.PrettyPrint( this );
}
```

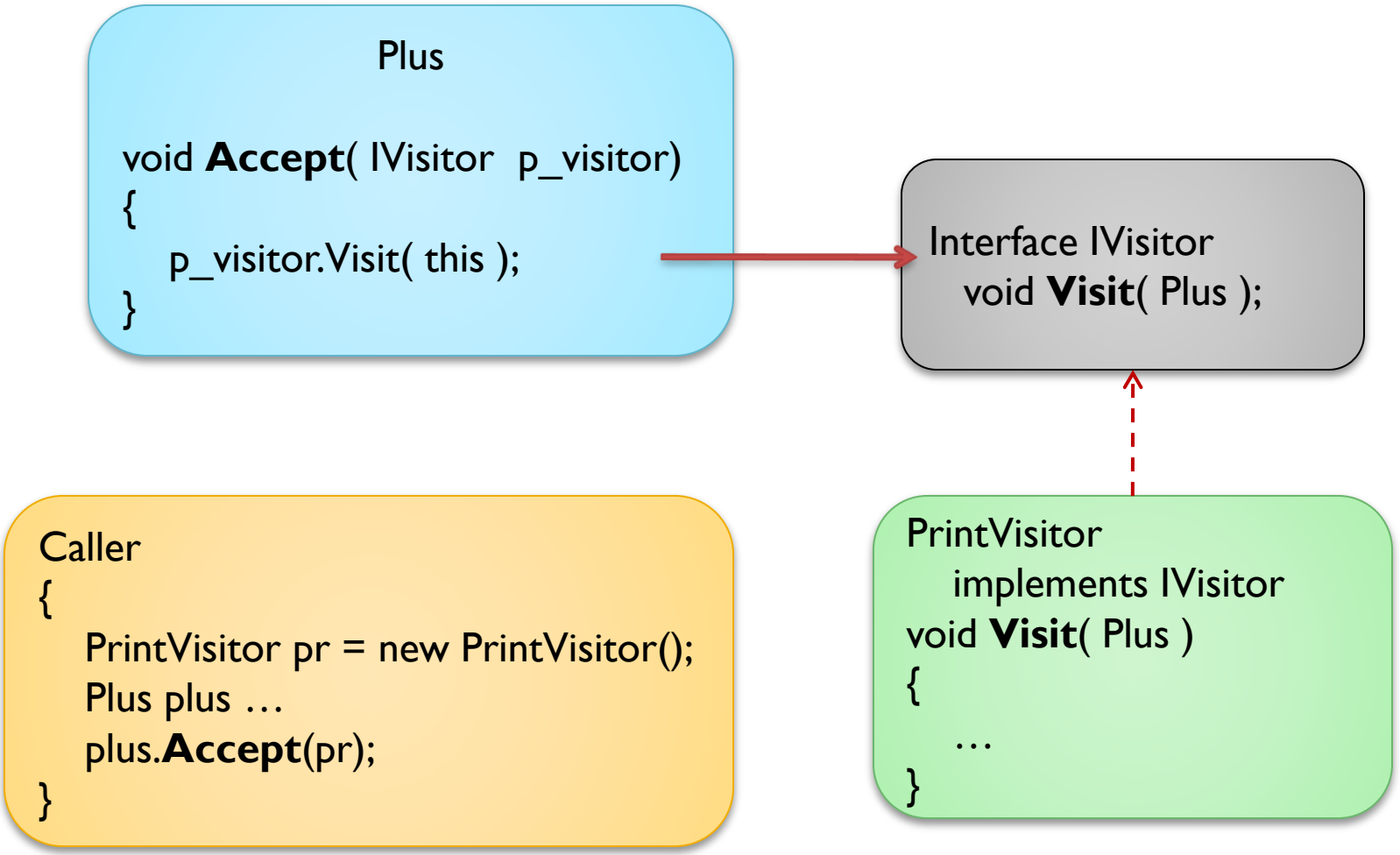
```
PrintVisitor

void PrettyPrint( Plus )
{
    …
}
```

- Client

```
Caller
{
    PrintVisitor pr = new PrintVisitor();
    Plus plus …
    plus.Print(pr);
}
```

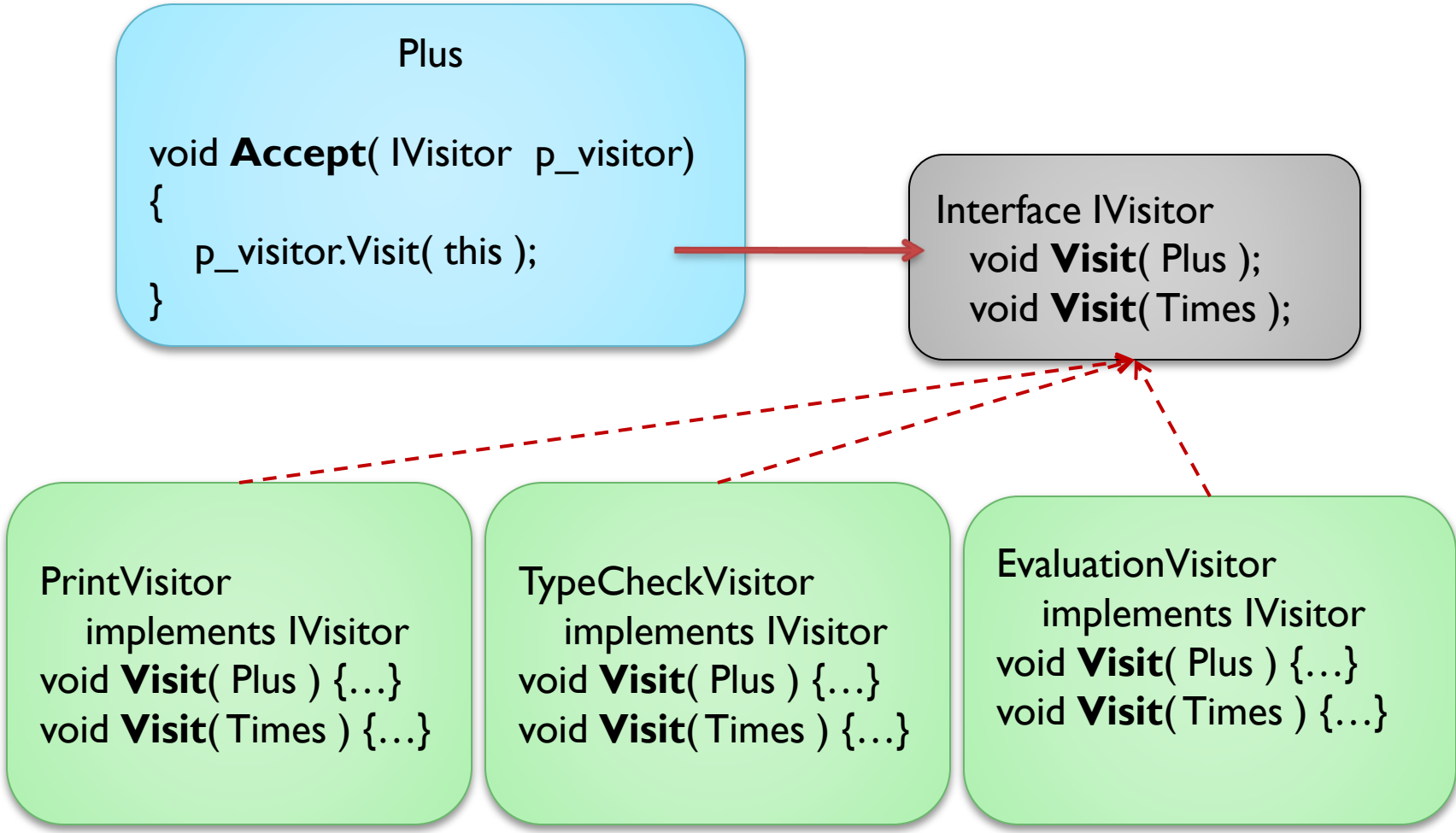
The Visitor Design Pattern

- Finally



The Visitor Design Pattern

- Benefits

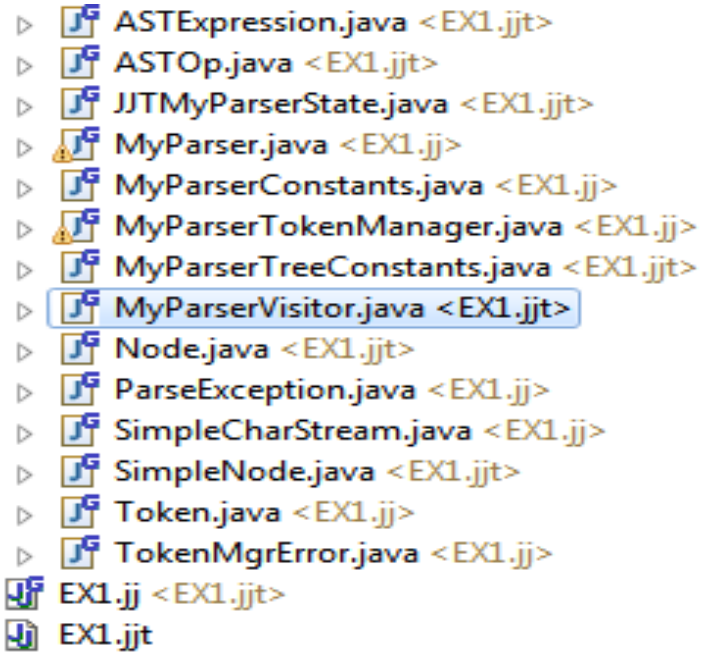


JJTree and the Visitor Pattern

```
Options
{
...
VISITOR=true;
...
}
```

Visitor interface

```
public interface MyParserVisitor
{

  public Object visit(SimpleNode node, Object data);

  public Object visit(ASTOp node, Object data);

  public Object visit(ASTExpression node, Object data);

}
```



Visitor - Node modification

```
Public class ASTExpression extends SimpleNode {
  public ASTExpression(int id) {
    super(id);
  }

  public ASTExpression(MyParser p, int id) {
    super(p, id);
  }

  /** Accept the visitor. **/
  public Object jjtAccept(MyParserVisitor visitor, Object data) {
    return visitor.visit(this, data);
  }
}
```

JJTree and the Visitor Pattern

```
Options
{
…
VISITOR=true;
VISITOR_DATA_TYPE="SymbolTable";
VISITOR_RETURN_TYPE="Object";
…
}
```

Semantic Analysis / Type Checking

- Consider the following if-statement

```
if ( Expression )
{
   …
}
```

Type Checking

- Replacing the expression

```
if ( 3 + 2 )
{
   …
}
```

- 3 + 2 results in an integer type whereas the if-statement is expecting a Boolean type

Example Grammar

INTEGER ::= DIGIT+

DIGIT ::= ["0"-"9"]

PLUS ::= "+“

MINUS ::= "-"

Number ::= INTEGER

Op ::= PLUS | MINUS

Expression ::= Number [Op Expression]

Type Checking

```
public class TypeChecker implements MyParserVisitor
{
    public boolean visit(SimpleNode node, Object data) { return false; }
    public boolean visit(ASTNumber node, Object data) {return true; }
    public boolean visit(ASTOp node, Object data) {return true; }

    public boolean visit(ASTExpression node, Object data)
    {
        int num = node.jjtGetNumChildren();
        if( ! (node.jjtGetChild(0) instanceof ASTNumber) )
            return false;
        if(num==1)
            return true;
        if(! node.jjtGetChild(1).jjtAccept(this, data) )
            return false;
        return node.jjtGetChild(2).jjtAccept(this, data);
    }
}
```

A better way – 1/5

- Create an enumerated type:

```
public enum DataType
{
    TypeUnknown,
    TypeError,
    TypeBoolean,
    TypeInteger,
    TypeOp,
    TypeCommand
}
```

A better way – 2/5

- Set the visitor return type to DataType:

```
options
{
   ...
   VISITOR=true;
   VISITOR_RETURN_TYPE="DataType";
   ...
}
```

A better way – 3/5

- Generate the visitor interface:

```
public interface MyParserVisitor
{
  public DataType visit(SimpleNode node, Object data);
  public DataType visit(ASTNumber node, Object data);
  public DataType visit(ASTOp node, Object data);
  public DataType visit(ASTExpression node, Object data);
}
```

A better way – 4/5

- Implement the visitor methods:

```
public class TypeChecker implements MyParserVisitor

public DataType visit(SimpleNode node, Object data) { return DataType.TypeError;}
public DataType visit(ASTNumber node, Object data) { return DataType.TypeInteger;}
public DataType visit(ASTOp node, Object data) { return DataType.TypeOp;}

public DataType visit(ASTExpression node, Object data) {
    int num = node.jjtGetNumChildren();
    if(node.jjtGetChild(0).jjtAccept(this, data) != DataType.TypeInteger)
        return DataType.TypeError;
    if(num==1)
        return DataType.TypeInteger;
    if(node.jjtGetChild(1).jjtAccept(this, data) != DataType.TypeOp)
        return DataType.TypeError;
    return node.jjtGetChild(2).jjtAccept(this, data);
}
```

A better way – 5/5

- Implement the call to the visitor:

```
Public class MyParser {
... main ...
Try
    {
      SimpleNode root = parser.Expression();

      MyParserVisitor visitor = new TypeChecker();
      DataType dataType = root.jjtAccept(visitor, null );

      System.out.println("DataType is " + dataType.toString());
...
```

Code Generation

- Traverse the tree and emit code for each of the visited nodes

- Sometimes, the emitted code depends on
  - a property of the node
  - Or properties of its children

Code Generation

- Similarly to Semantic Analysis, we can make use of the visitor pattern
- We are translating portions of the source language into snippets of the target language
- We have to take special care about the difference in the semantics of the languages involved

Code Generation

- On the next slide there is a sample grammar
- Our task is to translate the source language into a Java source file

Code Generation

Letter	::=	[“a“-“z“ “A“-“z“]
Digit	::=	[“0“-“9“]
Type	::=	“Numeric“	“Boolean“
Literal	::=	BooleanLiteral	NumericLiteral
BooleanLiteral	::=	“true”	“false”
NumericLiteral	::=	{ Digit+ } [“.” Digit+]
Identifier	::=	(_ | Letter) { _	Letter	Digit }
DeclarationStatement	::=	“var” Identifier “:” Type “=” Literal “;”
WriteStatement	::=	“write” Identifier “;”
Statement	::=	DeclarationStatement	WriteStatement
StatementBlock	::=	"{" { Statement } "}“
Header	::=	"Script" Identifier“;”
Program ::=	Header StatementBlock	

Code Generation

- What to look for
  - In the source grammar
    - Tokens
    - Production Rules
    - Types
  - In the target language
    - Equivalent Types in the target language
      - Some types might have to be constructed
    - Equivalent constructs

Code Generation

- In the general case:
  - Literals
  - Identifiers
  - Types / Type system
    - Conveys meaning
  - Expressions
    - Are the most important part of the language
  - Statements
    - Shape the Abstract Syntax Tree / Parse Tree

Code Generation - Visitor

```
public interface ICompilerVisitor
{
  public Object visit(SimpleNode node, SymbolTable data);
  public Object visit(Type node, SymbolTable data);
  public Object visit(Literal node, SymbolTable data);
  public Object visit(DeclarationStatement node, SymbolTable
    data);
  public Object visit(WriteStatement node, SymbolTable data);
  public Object visit(Header node, SymbolTable data);
  public Object visit(Program node, SymbolTable data);
}
```

Code Generation - Visitor

```
public class CodeGenerator
   implements ICompilerVisitor
{
   protected FileWriter filewriter;

   CodeGenerator() { … }
   CodeGenerator(FileWriter p_filewriter) { … }

   void Emit(String p_str) { … }
   void EmitLn(String p_str) { … }
   …
}
```

Code Generation

Letter	::=	[“a“-“z“ “A“-“z“]
Digit	::=	[“0“-“9“]
Type	::=	“Numeric”	“Boolean”
Literal	::=	BooleanLiteral	NumericLiteral
BooleanLiteral	::=	“true”	“false”
NumericLiteral	::=	{ Digit+ } [“.” Digit+]
Identifier	::=	(_ | Letter) { _	Letter	Digit }
DeclarationStatement	::=	“var” Identifier “:” Type “=” Literal “;”
WriteStatement	::=	“write” Identifier “;”
Statement	::=	DeclarationStatement	WriteStatement
StatementBlock	::=	“{” { Statement } “}”
Header	::=	“Script” Identifier“;”
Program	::=	Header StatementBlock

Code Generator

```
public Object visit(Header node, SymbolTable data)
{
    // Source: Script Identifier;
    // Target: public class "Identifier" { public static void main(args) { ... } }

    String scriptname =
    (String)(((Identifier)node.jjtGetChild(0)).jjtGetValue());

    Emit("public class " + scriptname + "{\r\n");
    Emit("public static void main(String args[]) {")

    node.jjtGetParent().jjtGetChild(1).jjtAccept(this,data);

    Emit("}");
    Emit("}");
}
```

Code Generator

```
public Object visit(WriteStatement node, SymbolTable data)
{
    // Source: "write" Identifier ";"
    // Target: System.out.print( Identifier );

    String identifiername =
    (String)(((Identifier)node.jjtGetChild(0)).jjtGetValue());

    Emit("System.out.print(" + identifiername + ");\r\n");
}
```

Code Generator

```
public Object visit(DeclarationStatement node, SymbolTable data)
{
    // Source :“var” Identifier “:” Type “=” Literal “;”
    // Target : Type identifier = literal;

    node.jjtGetChild(1)).jjtAccept(this, data);
    Emit( “ ” );

    String identifiername = (String)(((Identifier)node.jjtGetChild(0)).jjtGetValue());
    Emit(identifiername);

    Emit( “ = ” );
    node.jjtGetChild(2)).jjtAccept(this, data);
    Emit( “;” );
}
```

Questions?

The End