



## Creating Syntax Trees Using JJTree

---

- JJTree is a preprocessor for JavaCC that inserts parse tree building actions at various places in the JavaCC source.
- The output of JJTree is run through JavaCC to create the parser.
- By default JJTree generates code to construct parse tree nodes for each nonterminal in the language; this behavior can be modified so that some nonterminals do not have nodes generated, or so that a node is generated for a part of a production's expansion.
- JJTree defines a Java interface `Node` that all parse tree nodes must implement. The interface provides methods for operations such as setting the parent of the node, and for adding children and retrieving them.
- JJTree operates in one of two modes, *simple* and *multi*. In simple mode each parse tree node is of concrete type `SimpleNode`; in multi mode the type of the parse tree node is derived from the name of the node.

## Decorations

---

JJTree provides decorations for *definite* and *conditional* nodes.

- A definite node is constructed with a specific number of children. For example: `#ADefiniteNode(INTEGER EXPRESSION)`
- A conditional node is constructed if and only if its condition evaluates to true. For example: `#ConditionalNode(BOOLEAN EXPRESSION)`
- By default JJTree treats each nonterminal as a separate node and derives the name of the node from the name of its production. You can give it a different name with the following syntax:  

```
void P1() #MyNode : { ... } { ... }
```
- If you want to suppress the creation of a node for a production you can use the following syntax:  

```
void P2() #void : { ... } { ... }
```
- You can make this the default behavior for non-decorated nodes by using the `NODE_DEFAULT_VOID` option.
- The current node can be accessed using the identifier `jjtThis`

# Creating Syntax Trees for the Straight Line Programming Language

---

```

/*****
***** SECTION 1 - OPTIONS *****/
*****/
options { MULTI = true; NODE_DEFAULT_VOID = true; NODE_PREFIX = ""; }

/*****
***** SECTION 2 - USER CODE *****/
*****/

PARSER_BEGIN(SLPParser)

public class SLPParser
{
    public static void main(String args[])
    {
        SLPParser parser;
        if (args.length == 0)
        {
            System.out.println("SLP Parser: Reading from standard input . . .");
            parser = new SLPParser(System.in);
        }
        else if (args.length == 1)
        {
            System.out.println("SLP Parser: Reading from file " + args[0] + "
            try
            {
                parser = new SLPParser(new java.io.FileInputStream(args[0]));
            }
            catch (java.io.FileNotFoundException e)
            {
                System.out.println("SLP Parser: File " + args[0] + " not found
                return;
            }
        }
    }
}

```

# Creating Syntax Trees for the Straight Line Programming Language

---

```

else
{
    System.out.println("SLP Parser: Usage is one of.");
    System.out.println("      java SLPParser < inputfile
    System.out.println("OR");
    System.out.println("      java SLPParser inputfile")
    return;
}
try
{
    Stm p = parser.Prog();
    p.dump("");
}
catch (ParseException e)
{
    System.out.println(e.getMessage());
    System.out.println("SLP Parser: Encountered errors during p
PARSER_END(SLPParser)

```

# Creating Syntax Trees for the Straight Line Programming Language

---

```

/*****
***** SECTION 3 - TOKEN DEFINITIONS *****/
*****

TOKEN_MGR_DECLS :
{
    static int commentNesting=0;
}

SKIP : /** Ignoring spaces/tabs/newlines ***/
{
    " "
    | "\t"
    | "\n"
    | "\r"
    | "\f"
}

SKIP : /** COMMENTS */
{
    "/*" { commentNesting++; } : IN_COMMENT
}

<IN_COMMENT> SKIP :
{
    "/*" { commentNesting++; }
    | "*/" { commentNesting--;
            if (commentNesting == 0)
                SwitchTo(DEFAULT);
            }
    | <~[]>
}

```

# Creating Syntax Trees for the Straight Line Programming Language

---

```

TOKEN : /* Keywords and punctuation */
{
    < SEMIC : ";" >
    | < ASSIGN : ":@" >
    | < PRINT : "print" >
    | < LBR : "(" >
    | < RBR : ")" >
    | < COMMA : "," >
    | < PLUS_SIGN : "+" >
    | < MINUS_SIGN : "-" >
    | < MULT_SIGN : "*" >
    | < DIV_SIGN : "/" >
}

TOKEN : /* Numbers and identifiers */
{
    < NUM : (<DIGIT>)+ >
    | < #DIGIT : ["0" - "9"] >
    | < ID : (<LETTER>)+ >
    | < #LETTER : ["a" - "z", "A" - "Z"] >
}

TOKEN : /* Anything not recognised so far */
{
    < OTHER : ~[] >
}

```

# Creating Syntax Trees for the Straight Line Programming Language

---

```

/*****
 * SECTION 4 - THE GRAMMAR *
 *****/

Stm Prog() #Stm : {}
{
  Stm() <EOF>
  { return jjtThis; }
}

void Stm() #Stm : {}
{
  (SimpleStm() [<SEMIC> Stm() #CompoundStm] )
}

void SimpleStm() : {}
{
  (Ident() <ASSIGN> Exp()) #AssignStm
  | (<PRINT> <LBR> ExpList() <RBR>) #PrintStm
}

void Exp() #Exp :
{ int oper; }
{
  (SimpleExp() [oper = BinOp() Exp() { jjtThis.setOper(oper); } #OpExp] )
}

void SimpleExp() : {}
{
  IdExp()
  | NumExp()
  | (<LBR> Stm() <COMMA> Exp() <RBR>) #EseqExp
}

```

# Creating Syntax Trees for the Straight Line Programming Language

---

```

void Ident() : {}
{
  <ID>
}

void IdExp() #IdExp :
{ Token t; }
{
  t = <ID>
  { jjtThis.setName(t.image); }
}

void NumExp() #NumExp :
{ Token t; }
{
  t = <NUM>
  { jjtThis.setNum(Integer.parseInt(t.image)); }
}

void ExpList() #ExpList : {}
{
  (Exp() <COMMA> ExpList() #PairExpList | {} #LastExpList )
}

int BinOp() : {}
{
  <PLUS_SIGN> { return 1; }
  | <MINUS_SIGN> { return 2; }
  | <MULT_SIGN> { return 3; }
  | <DIV_SIGN> { return 4; }
}

```

## Tree Nodes

The implementation of some of the tree nodes are as follows:

```
public class NumExp extends SimpleNode {
    private int num;
    public void setNum(int n) { num = n; }
    public NumExp(int id) { super(id); }
    public NumExp(SLPParser p, int id) { super(p, id); }
}

public class IdExp extends SimpleNode {
    private String name;
    public void setName(String n) { name = n; }
    public IdExp(int id) { super(id); }
    public IdExp(SLPParser p, int id) { super(p, id); }
}

public class OpExp extends SimpleNode {
    private int oper;
    public void setOper(int o) { oper = o; }
    public OpExp(int id) { super(id); }
    public OpExp(SLPParser p, int id) { super(p, id); }
}
}
```

## Traversing Trees

- Want to write a function that calculates the value of an expression tree:

```
int Calculate(Exp tree, Table t) { ... }
```

- How do we determine what kind of expression we are traversing (identifier, number, sequence expression or binary expression)?

```
IntAndTable evaluate(Exp tree, Table t) {
    if (tree instanceof NumExp)
        return new IntAndTable(((NumExp)tree).num,t);
    else if (tree instanceof IdExp)
        return new IntAndTable(t.lookup(t,((IdExp)tree).name),t);
    else if (tree instanceof EseqExp)
        return evaluate((Exp)tree.getChild(1),interpret((String)
            tree // Binary Operator Expression
        )
        {
            IntAndTable arg1 = evaluate((Exp)tree.getChild(0),t);
            IntAndTable arg2 = evaluate((Exp)tree.getChild(1),arg1
            switch ((OpExp)tree.oper) {
                case 1: return new IntAndTable(arg1.i+arg2.i,arg2.t);
                case 2: return new IntAndTable(arg1.i-arg2.i,arg2.t);
                case 3: return new IntAndTable(arg1.i*arg2.i,arg2.t);
                case 4: return new IntAndTable(arg1.i/arg2.i,arg2.t);
            }
        }
}
```

## Using Instanceof and Type Casts

---

- The code constantly uses type casts and instanceof to determine what class of object it is considering.
- This is not object-oriented.
- There is a better way – the 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
  - 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

## Using the Visitor Pattern With JJTree

---

- JJTree provides some basic support for the visitor design pattern.
- If the VISITOR option is set to true JJTree will insert an jjtAccept() method into all of the node classes it generates.
- It also generates a visitor interface that can be implemented and passed to the nodes to accept:

```
public interface SLPParserVisitor
{
    public Object visit(SimpleNode node, Object data);
    public Object visit(Stm node, Object data);
    public Object visit(CompoundStm node, Object data);
    public Object visit(AssignStm node, Object data);
    public Object visit(PrintStm node, Object data);
    public Object visit(Exp node, Object data);
    public Object visit(OpExp node, Object data);
    public Object visit(EseqExp node, Object data);
    public Object visit(IdExp node, Object data);
    public Object visit(NumExp node, Object data);
    public Object visit(ExpList node, Object data);
    public Object visit(PairExpList node, Object data);
    public Object visit(LastExpList node, Object data);
}
```

## Visitor Implementation

---

```

public Object visit(NumExp n, Table t) {
    return new IntAndTable(n.num,t);
}

public Object visit(IdExp i,Table t) {
    return new IntAndTable(t.lookup(t,i.name),t);
}

public Object visit(EseqExp e, Table t) {
    Table t' = e.getChild(0).jttAccept(this,t);
    return e.getChild(1).jttAccept(this,t');
}

public Object visit(OpExp o, Table t) {
    IntAndTable arg1 = o.getChild(0).jttAccept(this,t);
    IntAndTable arg2 = o.getChild(1).jttAccept(this,arg1.t);
    switch (o.oper) {
        case 1: return new IntAndTable(arg1.i+arg2.i,arg2.t);
        case 2: return new IntAndTable(arg1.i-arg2.i,arg2.t);
        case 3: return new IntAndTable(arg1.i*arg2.i,arg2.t);
        case 4: return new IntAndTable(arg1.i/arg2.i,arg2.t);
    }
}

```

## Tree Nodes

---

The implementation of some of the tree nodes are as follows:

```

public class NumExp extends SimpleNode {
    private int num;
    public void setNum(int n) { num = n; }
    public NumExp(int id) { super(id); }
    public NumExp(SLPParser p, int id) { super(p, id); }
    public Object jttAccept(SLPParserVisitor visitor, Object data) {
        return visitor.visit(this, data);
    }
}

public class IdExp extends SimpleNode {
    private String name;
    public void setName(String n) { name = n; }
    public IdExp(int id) { super(id); }
    public IdExp(SLPParser p, int id) { super(p, id); }
    public Object jttAccept(SLPParserVisitor visitor, Object data) {
        return visitor.visit(this, data);
    }
}

public class OpExp extends SimpleNode {
    private int oper;
    public void setOper(int o) { oper = o; }
    public OpExp(int id) { super(id); }
    public OpExp(SLPParser p, int id) { super(p, id); }
    public Object jttAccept(SLPParserVisitor visitor, Object data) {
        return visitor.visit(this, data);
    }
}

```

- The control flow goes back and forth between the visit methods in the Visitor and the jttAccept methods in the object structure.

## Visitors

---

- Visitor makes adding new operations easy: simply write a new visitor.
- A visitor gathers related operations: it also separates unrelated ones.
- Adding new classes to the object structure is hard. Key consideration: are you most likely to change the algorithm applied over an object structure, or are you most likely to change the classes of objects that make up the structure?
- Visitor can break encapsulation: Visitor's approach assumes that the interface of the data structure classes is powerful enough to let visitors do their job. As a result, the pattern often forces you to provide public operations that access internal state, which may compromise its encapsulation.