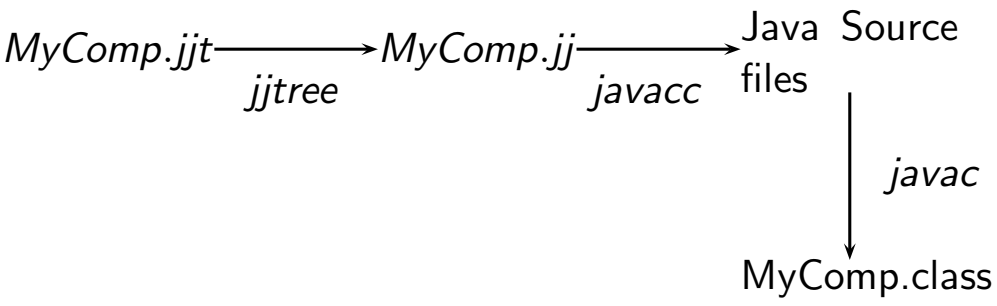
CA4003 - Compiler Construction

JJTree - A Brief Introduction

David Sinclair

JJTree

JJTree is a preprocessor for JavaCC that is used to build *abstract syntax trees*.



- The *.jjt* file is an extension of a JavaCC *.jj* file that contains annotations that helps build the *abstract syntax tree*.
- The resulting *.jj* file is processed by JavaCC.
- The user can edit the generated *.java* files and supply additional files to build the compiler.

Example

We will use the following example grammar to highlight the various aspects of JJTree. This grammar describes a simple language that consists of multiple statements (each ending in a semicolon). Each statement is ether a declaration of a variable or an expression. Variables can be integers or booleans, and expressions can use integer arithmetic operators or boolean operators.

Program	→	Statements <EOF>
Statements	→	SimpleStatement ';'
Statements	→	SimpleStatement ';' Statements
SimpleStatement	→	expr | declaration
declaration	→	(int | bool) Identifier
expr	→	Term (AddOp | BoolOp) Term
Term	→	(∼Factor) | (Factor MulOp Factor)
Factor	→	'(' expr ')' | Number | Identifier
MulOp	→	'*' | '/'
AddOp	→	'+' | '-'
BoolOp	→	'&' | '|'

Example (2)

We will use JJTree to build an *abstract syntax tree (AST)* for any given input using the previous grammar.

We will then analyse the AST to see if there are any incompatible types in each statement.

- Any addition, subtraction, multiplication or division that involves a boolean variable.
- Any logical conjunction, disjunction or negation that involves an integer variable.

In order to perform this analysis we need to record the type of every variable when it is declared. We will use a **very simple** implementation of a symbol table, which only records a symbols declaration. A proper symbol table would also have to deal with the scope and remove symbols from the symbol table as the scope changes.

Basic JJTree Operation

Whenever a production rule in the *.jjt* file is invoked, a node in the AST is created. The type of this node is derived from the name of the production rule with the prefix AST. This is the default behaviour which can be be overridden by the JJTree Options. By adding the `#void` decoration in the production rule declaration will prevent JJTree from creating the AST node.

In a production rule, if you add a decoration of the form `#Name(n)` in the semantic actions, JJTree will create a node of type `Name` with *n* children, e.g. `#Add(2)` creates an `Add` node with 2 children.

The identifier `jjtThis` refers to the current node during the generation of the AST.

JJTree Options

JJTree introduces some additional options over JavaCC. The most important ones are:

- **MULTI** When set to `false` all AST nodes are derived from the class `SimpleNode`. When set to true, a decoration `#Name` will generate a node derived from the class `ASTName`.
- **NODE_PREFIX** The default prefix in `MULTI` mode is `AST`, but this can be redefined.
- **VISITOR** When set to `true`, this will insert a `jjtAccept` method in each node class and generate a `Visitor` interface.
- **VISITOR_DATA_TYPE** By default, the `Visitor` interface receives a generic data object of the class `Object`. This can be specialised.
- **VISITOR_RETURN_TYPE** By default, the `Visitor` interface returns a generic data object of the class `Object`. This can be specialised.

JJTree AST Nodes

All AST nodes implement the `Node` interface, which has the following useful methods.

`public void jjtOpen();` and `public void jjtClose();`
You need to open a node before adding children to it and close it afterwards.

`public void jjtAddChild(Node n, int i);`
Add the node to the current node's list of children,

`public Node jjtGetChild(int i);`
Get the i-th child (numbering starts from 0),

`public int jjtGetNumChildren();`
Returns the current node's number of children,

`public void childrenAccept(MyParserVisitor visitor);`
When the `VISITOR` option is `true`, this walks over the children nodes in turn, asking them to accept the visitor.

ExprLang.jjt

```
/*******************************
 ***** SECTION 1 - OPTIONS *****
 *******************************/

options
{
  IGNORE_CASE = false;
  MULTI=true;
  VISITOR = true;
}

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

PARSER_BEGIN(ExprLang)

import java.io.*;
import java.util.*;

public class ExprLang
{
  public static Hashtable ST = new Hashtable();

  public static void main(String[] args) throws ParseException, FileNotFoundException
  {
    String temp;
    STC temp2;
```

ExprLang.jjt (2)

```
    if (args.length < 1)
    {
      System.out.println("Please pass in the filename.");
      System.exit(1);
    }

    ExprLang parser = new ExprLang(new FileInputStream(args[0]));

    SimpleNode root = parser.program();

    System.out.println();
    System.out.println("Program:");
    PrintVisitor pv = new PrintVisitor();
    root.jjtAccept(pv, null);

    System.out.println();
    System.out.println("Type Checking:");
    TypeCheckVisitor tc = new TypeCheckVisitor();
    root.jjtAccept(tc, ST);
  }
}

PARSER_END(ExprLang)
```


ExprLang.jjt (3)

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

TOKEN_MGR_DECLS:
{
  static int linenumber = 0;
}

SKIP:   /* Whitespace */
{
  "\t"
| "\n" {linenumber++;}
| "\r"
| " "
}

TOKEN:
{
  <LPAREN: "(">
| <RPAREN: ")">
| <ADD_OP: "+" | "-">
| <MULT_OP: "*" | "/">
| <NOT_OP: "~">
| <BOOL_OP: "&" | "|">
| <INT: "int">
| <BOOL: "bool">
| <NUMBER: (["0"-"9"])+>
| <ID: (["a"-"z","A"-"Z"])+>
| <SEMIC: ";">
}
```

ExprLang.jjt (4)

```
/**************************************************************************
 * SECTION 4 - THE GRAMMAR & PRODUCTION RULES - WOULD NORMALLY START HERE *
 **************************************************************************/

SimpleNode program() : {}
{
  Stms() <EOF> {return jjtThis;}
}

void Stms() #void : {}
{
  (SimpleStm() <SEMIC> [Stms() #Stms(2)] )
}

void SimpleStm() #void : {}
{
  (expression())
 | (declaration())
}

void declaration() #void : {Token t; String name;}
{
  t = <INT> name = identifier() {jjtThis.value = t.image;
                          ST.put(name, new STC("Int", name));} #Decl(1)
| t = <BOOL> name = identifier() {jjtThis.value = t.image;
                           ST.put(name, new STC("Bool", name));} #Decl(1)
}
```


ExprLang.jjt (5)

```
void expression() #void : {Token t;}
{
  term()
  (
    (t = <ADD_OP> term() {jjtThis.value = t.image;} #Add_op(2)
    )
  | (t = <BOOL_OP> term() {jjtThis.value = t.image;} #Bool_op(2)
    )
  )*
}

void term() #void : {Token t;}
{
  <NOT_OP> factor() #Not_op(1)
|
  factor()
  (t = <MULT_OP> factor() {jjtThis.value = t.image;} #Mult_op(2)
  )*
}

void factor() #void : {}
{
  (<LPAREN> expression() #Exp(1) <RPAREN>
  | number()
  | identifier()
  )
}
```

ExprLang.jjt (6)

```
void number() : {Token t;}
{
  t = <NUMBER> {jjtThis.value = t.image;}
}

String identifier() : {Token t;}
{
  t = <ID> {jjtThis.value = t.image; return t.image;}
}
```


STC.java

Our very basic symbol table implementation.

```
import java.util.*;

public class STC extends Object
{
    String type;
    String value;

    public STC(String itype, String ivalue)
    {
         type = itype;
         value = ivalue;
    }
}
```

Visitors

In this simple example, we are implementing two visitors:

1. A printing visitor
2. A type checking visitor

Each visitor must implement the `ExprLangVisitor` interface that was generated by JJTree.

Since our `.jjt` file contained decorations to generate nodes of the type `Decl`, `Stms`, `Add_op`, `Bool_op`, `Mult_op`, `Not_op` and `Exp`, the `ExprLangVisitor` interface needs to handle nodes of type `ASTprogram`, `ASTDecl`, `ASTStms`, `ASTAdd_op`, `ASTBool_op`, `ASTMult_op`, `ASTNot_op`, `ASTExp`, `ASTidentifier` and `ASTnumber`.

- Where did `ASTprogram`, `ASTidentifier` and `ASTnumber` come from?

PrintVisitor.java

```
public class PrintVisitor implements ExprLangVisitor
{
    public Object visit(SimpleNode node, Object data)
    {
         throw new RuntimeException("Visit SimpleNode");
    }

    public Object visit(ASTprogram node, Object data)
    {
         node.jjtGetChild(0).jjtAccept(this, data);
         System.out.println(";");
         return(data);
    }

    public Object visit(ASTDecl node, Object data)
    {
         System.out.print(node.value + " ");
         node.jjtGetChild(0).jjtAccept(this, data);
         return data;
    }

    public Object visit(ASTStms node, Object data)
    {
         node.jjtGetChild(0).jjtAccept(this, data);
         System.out.println(";");
         node.jjtGetChild(1).jjtAccept(this, data);
         return data;
    }
```

PrintVisitor.java (2)

```
    public Object visit(ASTAdd_op node, Object data)
    {
          node.jjtGetChild(0).jjtAccept(this, data);
          System.out.print(" " + node.value + " ");
          node.jjtGetChild(1).jjtAccept(this, data);
          return data;
    }

    public Object visit(ASTBool_op node, Object data)
    {
         node.jjtGetChild(0).jjtAccept(this, data);
         System.out.print(" " + node.value + " ");
         node.jjtGetChild(1).jjtAccept(this, data);
         return data;
    }

    public Object visit(ASTMult_op node, Object data)
    {
         node.jjtGetChild(0).jjtAccept(this, data);
         System.out.print(" " + node.value + " ");
         node.jjtGetChild(1).jjtAccept(this, data);
         return data;
    }

    public Object visit(ASTNot_op node, Object data)
    {
         System.out.print("~");
         return(node.jjtGetChild(0).jjtAccept(this, data));
    }
```


PrintVisitor.java (3)

```
    public Object visit(ASTExp node, Object data)
    {
         System.out.print("(");
         node.jjtGetChild(0).jjtAccept(this, data);
         System.out.print(")");
         return(data);
    }

    public Object visit(ASTidentifier node, Object data)
    {
         System.out.print(node.value);
         return data;
    }

    public Object visit(ASTnumber node, Object data)
    {
         System.out.print(node.value);
         return data;
    }
}
```

TypeCheckVisitor.java

The `TypeCheckVisitor` checks each expression to see if the operands are of the correct type for each operator. When it finds a type violation it uses the `PrintVisitor` to display the offending expression.

```
import java.util.*;

public class TypeCheckVisitor implements ExprLangVisitor
{
    public Object visit(SimpleNode node, Object data)
    {
         throw new RuntimeException("Visit SimpleNode");
    }

    public Object visit(ASTprogram node, Object data)
    {
         node.jjtGetChild(0).jjtAccept(this, data);
         return DataType.Program;
    }

    public Object visit(ASTDecl node, Object data)
    {
         return DataType.Declaration;
    }
```


TypeCheckVisitor.java (2)

```
    public Object visit(ASTStms node, Object data)
    {
         PrintVisitor pv = new PrintVisitor();

         if ((DataType)node.jjtGetChild(0).jjtAccept(this, data) == DataType.TypeUnknown)
         {
           System.out.print("Type error: ");
           node.jjtGetChild(0).jjtAccept(pv, null);
           System.out.println();
         }

         return (node.jjtGetChild(1).jjtAccept(this, data));
    }

    public Object visit(ASTAdd_op node, Object data)
    {
         if (((DataType)node.jjtGetChild(0).jjtAccept(this, data) == DataType.TypeInteger)
            && ((DataType)node.jjtGetChild(1).jjtAccept(this, data) == DataType.TypeInteger))
             return DataType.TypeInteger;
         else
             return DataType.TypeUnknown;
    }

    public Object visit(ASTBool_op node, Object data)
    {
         if (((DataType)node.jjtGetChild(0).jjtAccept(this, data) == DataType.TypeBoolean)
            && ((DataType)node.jjtGetChild(1).jjtAccept(this, data) == DataType.TypeBoolean))
             return DataType.TypeBoolean;
         else
             return DataType.TypeUnknown;
    }
```

TypeCheckVisitor.java (3)

```
    public Object visit(ASTMult_op node, Object data)
    {
         if (((DataType)node.jjtGetChild(0).jjtAccept(this, data) == DataType.TypeInteger)
            && ((DataType)node.jjtGetChild(1).jjtAccept(this, data) == DataType.TypeInteger))
             return DataType.TypeInteger;
         else
             return DataType.TypeUnknown;
    }

    public Object visit(ASTNot_op node, Object data)
    {
         if ((DataType)node.jjtGetChild(0).jjtAccept(this, data) != DataType.TypeBoolean)
             return DataType.TypeUnknown;
         else
            return DataType.TypeBoolean;
    }

    public Object visit(ASTExp node, Object data)
    {
        return(node.jjtGetChild(0).jjtAccept(this, data));
    }
```


TypeCheckVisitor.java (4)

```
    public Object visit(ASTidentifier node, Object data)
    {
         Hashtable ST = (Hashtable) data;
         STC hashTableEntry;

         hashTableEntry = (STC)ST.get(node.value);
         if (hashTableEntry.type == "Int")
         {
             return DataType.TypeInteger;
         }
         else if (hashTableEntry.type == "Bool")
         {
             return DataType.TypeBoolean;
         }
        else
        {
            return DataType.TypeUnknown;
         }
    }

    public Object visit(ASTnumber node, Object data)
    {
         return DataType.TypeInteger;
    }
}
```

DataType.java

```
public enum DataType
{
    Program,
    Declaration,
    TypeUnknown,
    TypeInteger,
    TypeBoolean
}
```