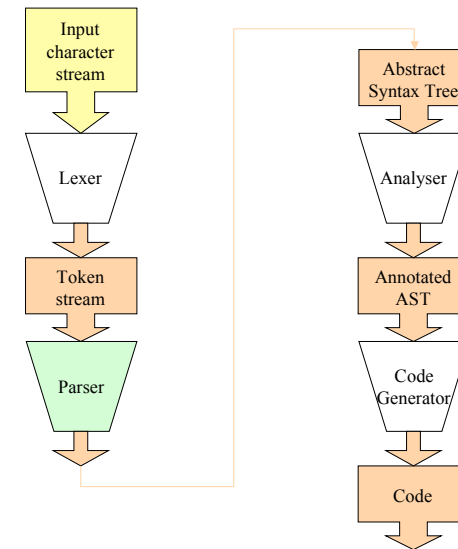


JJTree

An easier way to create an
Abstract Syntax Tree

The Compilation Task



Automated?

- The initial stages of compilation are very mechanistic
 - Lexer
 - Text to stream of tokens
 - Pattern matching
 - Parser
 - Application of grammar rules
 - Recovery from errors
- ⇒ AST Generation
 - Data structure for subsequent stages

JJTree

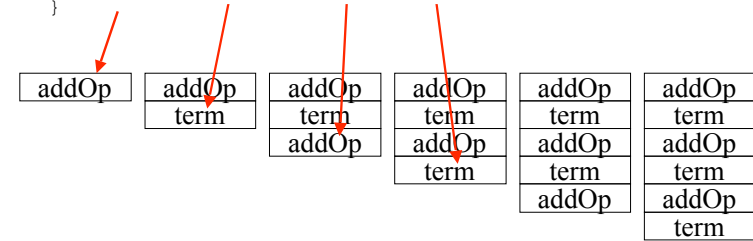
- Preprocessor to JavaCC
 - Generates .jj file that includes code to build an AST
- JJTree generates code to construct parse tree nodes for each nonterminal in the language.
 - This behaviour can be modified
 - some nonterminals do not have nodes generated,
 - or a node is generated for a part of a production's expansion

JJTree

- JJTree constructs the parse tree from the bottom up
 - Uses a stack to push created nodes
 - When a parent node is created
 - it pops the children from the stack
 - adds them to the parent
 - and pushes the new parent node on the stack
 - The stack can be manipulated by user code

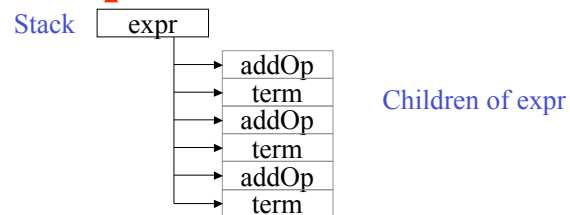
Bottom up AST Generation

- `-2+p("ok",3/4)-4;`
- ```
void expr() : {}
{
 [addOp()] term() (addOp() term()) *
}
```



# Bottom up AST Generation

- `-2+p("ok",3/4)-4;`
- ```
void expr() : {}
{
  [addOp()] term() ( addOp() term() ) *
}
```



Modes

- Simple
 - each parse tree node is an instance of SimpleNode
- Multi
 - in multi mode the type of the parse tree node is derived from the name of the node.
 - If you don't provide an implementation of a node class JJTree will generate a sample implementation.



Example 1

- Only modify initial production

```
– from
void start() : {}
{
    expr() ";"
}
– to
SimpleNode start() : {}
{
    expr() ";"
    { return jjtThis; }
}
```

Example 1

- Generates the following AST where all nodes are instances of SimpleNode, eg. $-2+p(\text{"ok"},3/4)-4$

```
start
  expr
    addOp
      term
        factor
          primary
            number 
        addOp
          term
            factor
              primary
                ident 
            ...
```

Creating Nodes

- A definite node
 - constructed with a specific number of children.
 - That many nodes are popped from the stack and made the children of the new node, which is then pushed on the stack itself.
 - #ADefiniteNode(INTEGER EXPRESSION)

Creating Nodes

- A conditional node
 - is constructed with all of the children on the stack if its condition evaluates to true.
 - If it evaluates to false, the node is not constructed.
 - #ConditionalNode(BOOLEAN EXPRESSION)

Creating Nodes

- By default JJTree treats each nonterminal as an indefinite 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 : { ... } { ... }

Example 2

- Create specific node classes for literals, eg. ident

```
public class ASTident extends SimpleNode {
    private String text;
    public ASTident(int id) {
        super(id);
    }
    public ASTident(Expression p, int id) {
        super(p, id);
    }
    public void setText(String n) {
        text = n;
    }
    public String toString() {
        return "ident: " + text;
    }
}
```

Annotations:

- Instance variable (ident text)
- Standard Constructors
- Accessor method
- To complete the job

Example 2

- Modify .jjt file to create/allow nodes for each non terminal production

```
options {
    MULTI=true;
}
```

- And to capture literals

```
void ident() :
{ Token t; }
{
    t=<IDENTIFIER>
    {
        jjtThis.setText(t.image);
    }
}
```

Implicitly refers to specific active node (ASTident)

Example 2

- Generates the following AST where all nodes are instances of Unique classes extending SimpleNode, eg.-
2+p("ok",3/4)-4

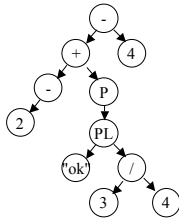
```
start          rvalue
expr           expr
addOp          term
term           factor
factor         primary
primary        number: 3
number: 2      multop
addOp          factor
term           primary
factor         number: 4
primary        addOp
ident: p       term
rvalue        factor
string: "ok"   primary
               number: 4
```

Example 3

- Generates a "useful" AST where all nodes are instances of unique classes extending SimpleNode, eg. $-2+p("ok",3/4)-4$

```

start
MinusNode
AddNode
  MinusNode
    number: 2
    ident: p
    parameterList
      string: "ok"
  DivideNode
    number: 3
    number: 4
  number: 4
  
```



Example 3

- The task is to only generate nodes where appropriate and generate an AST with a suitable shape.
- Remove superfluous intermediate nodes

```

void rvalue() #void : {}
{
  expr() | string()
}
  
```

Does not
generate a node

Example 3

- Generate only appropriate nodes

```

void addOp() #void : {}
{
  "+" #AddNode | "-" #MinusNode
}
  
```

Does not generate an
addOp node

But does generate either
an AddNode or
MinusNode

Example 3

- Reorder nodes

```

void expr() #void :
{
  Node unaryOp = null;
}
{
  [addOp() {unaryOp = jjtree.popNode();}] term() {
    if (unaryOp != null) {
      unaryOp.jjtAddChild(jjtree.popNode(), 0);
      jjtree.pushNode(unaryOp);
    }
  } ( addOp() term() {
    Node n1,n2,op;
    n2 = jjtree.popNode();
    n1 = jjtree.popNode();
    op.jjtAddChild(n1,0);
    op.jjtAddChild(n2,1);
    jjtree.pushNode(op);
  } ) *
}
  
```

If a unary operation exists
make the following term a
child of it

Restructure tree with a
binary operator having
exactly 2 children

More features of JJTree

- Visitor design pattern support
 - JJTree can insert an `jjtAccept()` method into all of the node classes it generates, and also generate a visitor interface

Questions

- How would a pretty printer be implemented using the visitor design pattern generation of JJTree?