The following slides show how to print a file after it has been treated by the compiler

The tokens are stored in a list

(See also the special tokens in a previous slide)

Returning the first token of the list (first solution)

```
Token rootProduction() throws Exception :
{Token first;}
{
    {first = getToken(1);}
    command()
    {return first;}
}
void command() throws Exception :
{}
{
    tag()  (command())*  endTag()
}
```

// can also be retrieved from the first node,
// as shown on one of the following pages

Printing all the tokens (not the special ones): in main()

```
      . . .
Token first = parser.rootProduction();
while (first != null) {
  System.out.println(first.image);
  first = first.next;
}
      . . .
```

Printing the special tokens

```
// To be called before every true token
// It inverts the order of the special tokens

static void followSpecialTokens (Token t){
        if (t == null)  return;
        followSpecialTokens(t.specialToken);
        System.out.print(t.image);
}
```

jjtree: creating a tree automatically (AST-abstract syntax tree)

Take the same file as before, but change extension to *.jjt* and call

jjtree Xxxx.jjt

You obtain *Xxxx.jj* and a list of classes. Call

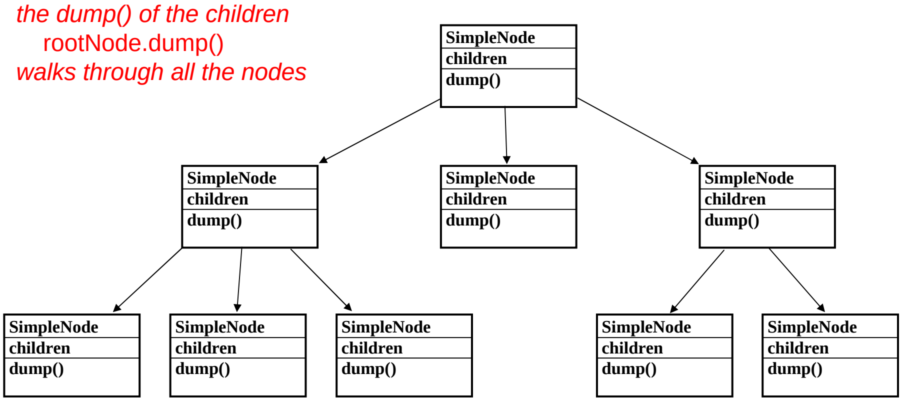
javacc Xxxx.jj

Compile with *javacc* and execute as usual, the tree is created, but of course the program does not use it yet.

// Source available in project Patterns, package tree, ParserDump.jjt

Tree: a node for each production (which node is generated can be controlled)

dump() is recursive, it calls the dump() of the children
rootNode.dump()
walks through all the nodes



How to link the tokens to the nodes

With the options:

```
NODE_SCOPE_HOOK=true;
```

The compiler *jjtree* enters the following commands at the beginning and the end of the every productions:

```
jjtree.openNodeScope(jjtn000);
    . . .
jjtreeCloseNodeScope(jjtn000);
```

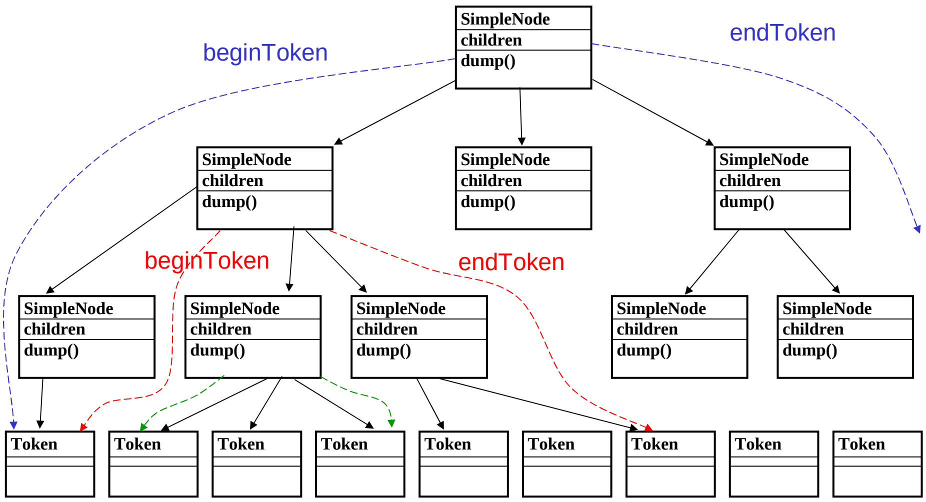
Add this in file *.jjt*

```
static void jjtreeOpenNodeScope(Node node) {
  ((SimpleNode)node).setBeginToken(getToken(1));
}
static void jjtreeCloseNodeScope(Node node) {
  ((SimpleNode)node).setEndToken(getToken(0));
}
```

… and this in SimpleNode.java

```java
private Token beginToken;
private Token endToken;
public Token getBeginToken() { return beginToken; }
public void setBeginToken(Token beginToken) {
        this.beginToken = beginToken;
}
public Token getEndToken() { return endToken; }
public void setEndToken(Token endToken) {
        this.endToken = endToken;
}
```

… then you obtain the following structure (Design Pattern!)



JavaCC: Controlling which node is generated

```
options {
      Node_DEFAULT_VOID=true;
      MULTI=true;
      NODE_PREFIX="EPFL_"
}
expression()   #Xxxx:   // creates a node with
{ }                     // name EPFL_Xxxx
{
      multExpression()
      (  "+"  multExpression()
      |  "-"  multExpression()
      )*
}
```

Getting the first token of the list (second solution)

The first token can also be retrieved from the first node:

```
node = parser.rootProduction()
```

where

```
SimpleNode rootProduction() : { }
    {
                 . . .
          {return jjtn000;}
    }
```

Printing the tokens covered by each node

```
public void dump(String prefix) {
      Token t = getBeginToken();
      Token te = getEndToken();
      System.out.print(toString(prefix) + "  !");
      while (t != null) {
          System.out.print(t.image);
          if (t == te)
              break;
          t = t.next;
        }
        System.out.println("!");
        // writes marks that represent the tree
          . . .
```

Printing the tokens covered by each node

```
public void dump(String prefix) {
                . . .
  System.out.println("!");   // create lines that
  if (children != null) {    // highlight the tree
    for (int i = 0; i < children.length; ++i) {
      SimpleNode n = (SimpleNode) children[i];
      if (n != null) {
        if (i != (children.length - 1))
          n.dump(prefix + "!");  // recursion
        else
          n.dump(prefix + " ");
} } } }
```

Visitor Design Pattern (GoF) handled by javacc

Used to perform work on the AST (abstract syntax tree).

The tree is walked through and the code of a ***visitor*** object is executed within each node.

Different visitors can perform different tasks (type checking, code generation) over the same tree, one after the other.

In the following, the example of the expressions interpreter will be handled with the help of the visitor pattern

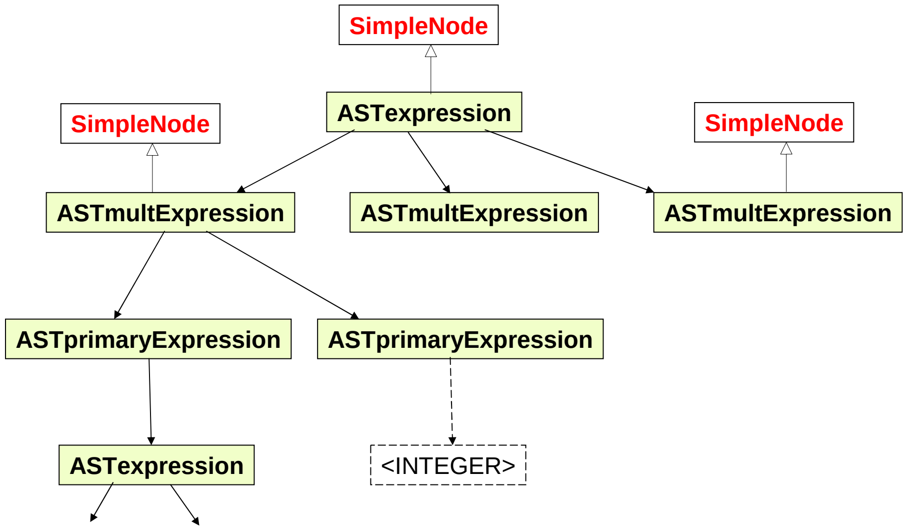
The nodes in the tree of the previous example

```
void expression() throws Exception :
{ }
{     multExpression()
       ( "+"  multExpression()
       | "-"  multExpression()
       )*
}
void multExpression() throws Exception :
{ }
{     primaryExpression()
      ( "*"   primaryExpression()   )*
}
void primaryExpression() throws Exception :
{ }
{
      (  <INTEGER>
      |   "(" expression() ")"
      )
}
```

Nodes created :

ASTexpression
ASTmultExpression
ASTprimaryExpression

The nodes of the tree of the previous example



Visitor Pattern: each node accepts a visitor (located in SimpleNode, see previous slide))

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

The call is forwarded to the ***visitor***, which contains one method per node. The method is identified by its argument. It receives this to allow the visitor to create the task determined by the node. Thus, ***different visitors*** can be executed with the same structure.

Visitor Pattern: the visitor contains a method for each kind of node

public class MyVisitor **implements** MyExpressionsVisitor {
  public Object visit (***ASTexpression*** node, Object data)
    { *your code* }
  public Object visit (***ASTmultExpression*** node, Object data)
    { *your code* }
  public Object visit (***ASTprimaryExpression*** node, Object data)
    { *your code* }
}

// MyExpressionsVisitor is generated automatically

// each task requires only a class like this one (no changes in the parser)

Visitor Pattern: in the main

```
// visit the nodes
node.childrenAccept(new MyVisitor(), null);
```

Visitor Pattern: recursive walk through the nodes (located in *SimpleNode*)

```
/** Accept the visitor. * */
public Object childrenAccept ( MyExpressionsVisitor visitor,
                               Object data) {
  if (children != null) {
    for (int i = 0; i < children.length; ++i) {
      ((SimpleNode)children[i]).childrenAccept(visitor, data);
    }
  }
  jjtAccept(visitor, data);
  return data;
}
```

These 2 lines had to be modified

Example of visitor

```
public Object visit(ASTprimaryExpression node, Object data) {
  if ( ((SimpleNode) node).jjtGetNumChildren() == 0 ) {
    node.ae = new Terminal(Integer.parseInt(node.getBeginToken().image));
  } else {
    node.ae = ((SimpleNode) node.jjtGetChild(0)).ae;
  }
  return null;
}
```

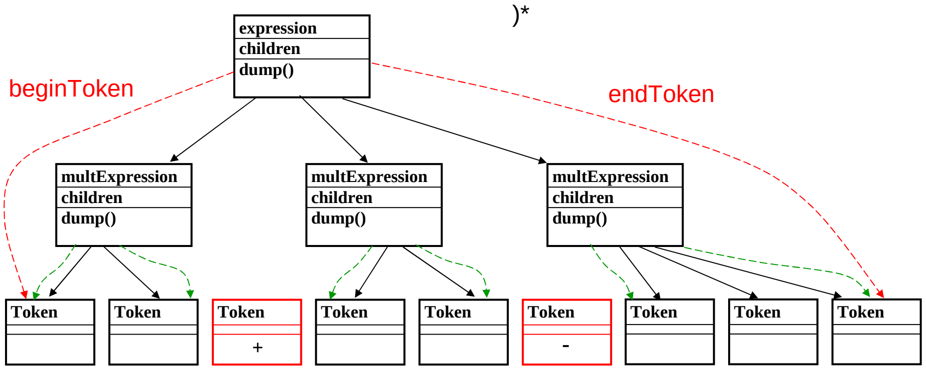
Either create a ***Terminal node*** { found <Integer>, namely no child }

or ***pass the expression*** found in the subtree { found "(" expression() ")" }

A field *AbstractExpression* has been defined in *SimpleNode*

Example of visitor (2)
(how to check if it is a “+” or a “-”)

```
expression:
      multExpression()
      (  "+“  multExpression()
      |  "-"   multExpression()
      )*
```



Another example of visitor (2)

```
public Object visit  (ASTexpression node, Object data) {
   if ( ((SimpleNode) node).jjtGetNumChildren() > 0 ) {
      SimpleNode nodeX = ((SimpleNode) node.jjtGetChild(0));
      node.ae = nodeX.ae;                                    // pass the subexpression
      int operation = nodeX.getEndToken().next.kind;  // previous operation
      for (int i = 1; i < ((SimpleNode) node).jjtGetNumChildren(); i++) {
         nodeX = (SimpleNode) node.jjtGetChild(i);
         if (operation == MyExpressionsConstants.PLUS) {
            node.ae = new PlusExpression(node.ae, nodeX.ae);
         } else {
            node.ae = new MinusExpression(node.ae, nodeX.ae);
         }
         operation = nodeX.getEndToken().next.kind;
   }  }
   return null;
}
```

→ *gets the token to see if it is a “+” or < “-” (next slide)*

In summary

ParserVisitor.jjt	previous *.jj* file complemented with the definition of *jjtreeOpenNodeScope*
SimpleNode.java	generated node complemented with the handling of *beginNode* and *endNode,* as well as an attribute for *an AbstractExpression*
MyVisitor.java	one method per node to generate the interpreter tree; generated entirely by the developer
AbstractExpression.java	classes for the Interpreter Pattern used
MinusExpression.java	in previous slides; the ***visitor*** creates them and
MultExpression.java	embed them within an interpreter tree
PlusExpression.java	
Terminal.java	
Stack.java	

Source available in project Patterns, package visitor

Example of the use of JavaCC: Synchronous Javascript

http://ltiwww.epfl.ch/~petitp/GenieLogiciel/test_s.html

Example of sJavascript

```
process Consumer(name) {
 var count
 this.put = function(counter) {      // a method
   count = counter
 }
 this.run = function () {            // the thread method
   for (;;) {
   accept put
   document.getElementById("consDisplay").innerHTML = count
}  }  }
```

Example of sJavascript

```
process Producer(name) {
 var ON = true
 var counter = 0
 this.run = function() {
   for (;;) {
     select {
     case
       startStop.clicked()
       ON = !ON
       document.getElementById("startStop")
                       .style.backgroundColor = ON?"green":"red"
     case
       when (ON)
       waituntil(now()+1000)
       counter++
       consumer.put(counter)
} } } }
```

scheduler

```
// sJavascript
process Consumer(name) {
   var count
   this.put = function(counter) {
      count = counter
   }
   this.run = function () {
      for (;;) {
         accept put
         document.getElementById
           ("consDisplay")
           .innerHTML = count
} } }
```

```
// standard Javascript
extending(Consumer, TaskControlBlock)
function Consumer(name) {
   Consumer.baseConstructor.call(this, name)
   var count
   this.$JM_put = 1
   this.put = function(counter) {
      count = counter
   }
   this.$J_state = 10000
   this.run = function () {
      for (;;) {
         switch(this.$J_state) {
         case 10000:
            this.$J_init()
            this.$J_accept(10002, this.$JM_put)
            this.$J_syncWait(29)
            return
         case 10002:
            document.getElementById
              ("consDisplay").innerHTML = count
            this.$J_state = 10000;
} } } }
```

sJavascript

The transformation on the last page has been performed with the Visitor Design Pattern