Session Plan

- Session 4a: ***More parsing and abstract syntax***
  - MiniJava introduction and parsing
  - lookahead
  - JavaCC grammars and semantic actions and values
  - simple expression evaluator
  - abstract syntax trees
  - using semantic actions to build abstract syntax trees
  - interpreting the trees
  - Visitors

MiniJava

- a subset of Java – example program:

```
class Factorial {
    public static void main(String[] a) {
        System.out.println(new Fac().ComputeFac(10));
    }
}

class Fac {
    public int ComputeFac(int num) {
        int num_aux ;
        if (num < 1)
            num_aux = 1 ;
        else
            num_aux = num * (this.ComputeFac(num-1)) ;
        return num_aux ;
    }
}
```

MiniJava Grammar I

Program → *MainClass ClassDecl **

MainClass → **class** *id* { **public static void main** (**String** [] *id*)
{ *Statement* } }

ClassDecl → **class** *id* { *VarDecl * MethodDecl ** }
→ **class** *id* **extends** *id* { *VarDecl* MethodDecl ** }

VarDecl → *Type id ;*

MethodDecl → **public** *Type id* (*FormalList*)
{ *VarDecl * Statement** **return** *Exp* ; }

MiniJava Grammar II

FormalList → *Type* *id* *FormalRest* *

→

FormalRest → , *Type* *id*

Type → **int []**

→ **boolean**

→ **int**

→ *id*

MiniJava Grammar III

Statement	→	{ *Statement* * }
	→	**if** (*Exp*) *Statement* **else** *Statement*
	→	**while** (*Exp*) *Statement*
	→	**System.out.println** (*Exp*) ;
	→	*id* **=** *Exp* **;**
	→	*id* [*Exp*] **=** *Exp* **;**
ExpList	→	*Exp* *ExpRest* *
	→	
ExpRest	→	**,** *Exp*

MiniJava Grammar IV

Exp → *Exp op Exp* **`&& < + - *`**

→ *Exp* **[** *Exp* **]**

→ *Exp* . **length**

→ *Exp* . *Id* **(** *ExpList* **)**

→ *INTEGER_LITERAL*

→ **true**

→ **false**

→ *id*

→ **this**

ambiguous?

→ **new int [** *Exp* **]**

→ **new** *id* **( )**

→ **!** *Exp*

→ **(** *Exp* **)**

MiniJava JavaCC grammar example

Program → *MainClass ClassDecl* *

MainClass → **class** *id* { **public static void main** (**String** [] *id*)
{ *Statement* } }

```
void Goal() :
{}
{
  MainClass()
  ( ClassDeclaration() )*
  <EOF>
}

void MainClass() :
{}
{
  "class" Identifier() "{"
    "public" "static" "void" "main" "(" "String" "[" "]" Identifier() ")"
    "{" Statement() "}"
  "}"
}
```

Local Lookahead

Statement → { *Statement* * }
→ **if** (*Exp*) *Statement* **else** *Statement*
→ **while** (*Exp*) *Statement*
→ **System.out.println** (*Exp*) ;
→ *id* **=** *Exp* **;**
→ *id* [*Exp*] **=** *Exp* **;**

```
void Statement() :
{}
{
  Block()
|
  LOOKAHEAD(2)
  AssignmentStatement()
|
  LOOKAHEAD(2)
  ArrayAssignmentStatement()
| …
```

```
void AssignmentStatement() :
{}
{
  Identifier() "=" Expression() ";"
}

void ArrayAssignmentStatement() :
{}
{
  Identifier() "[" Expression() "]" "=" Expression() "
}
```

Syntactic lookahead

Exp	→	*Exp* && *Exp*
		…
	→	*Exp* **[** *Exp* **]**
	→	*Exp* . **length**
	→	*Exp* . *Id* **(** *ExpList* **)**

```
void Expression() :
{}
{
  LOOKAHEAD( PrimaryExpression() "&&" )
  AndExpression()
|
…
  LOOKAHEAD( PrimaryExpression() "[" )
  ArrayLookup()
|
  LOOKAHEAD( PrimaryExpression() "." "length" )
  ArrayLength()
|
  LOOKAHEAD( PrimaryExpression() "." Identifier() "(" )
  MethodCall()
|
  PrimaryExpression()
}
```

Syntactic lookahead

```
void AndExpression() :
{}
{
  PrimaryExpression() "&&" PrimaryExpression()
}

void ArrayLength() :
{}
{
  PrimaryExpression() "." "length"
}

void MethodCall() :
{}
{
  PrimaryExpression() "." Identifier()
  "(" ( ExpressionList() )? ")"
}
```

```
void PrimaryExpression() :
{}
{
  IntegerLiteral()
|
  TrueLiteral()
|
  FalseLiteral()
|
  Identifier()
| …
```

Semantic actions

- each terminal and non-terminal associated with own type of semantic value
- terminal (token) semantic values are the tokens returned by the lexical analyser (type Token in JavaCC)
- non-terminals semantic values are given depending on what you want the rules to do
- semantic action for rule $A \rightarrow B\ C\ D$
  – returns type associated with A
  – can build this from values associated with B, C, D
- JavaCC allows us to intersperse actions within rules (written in {…})

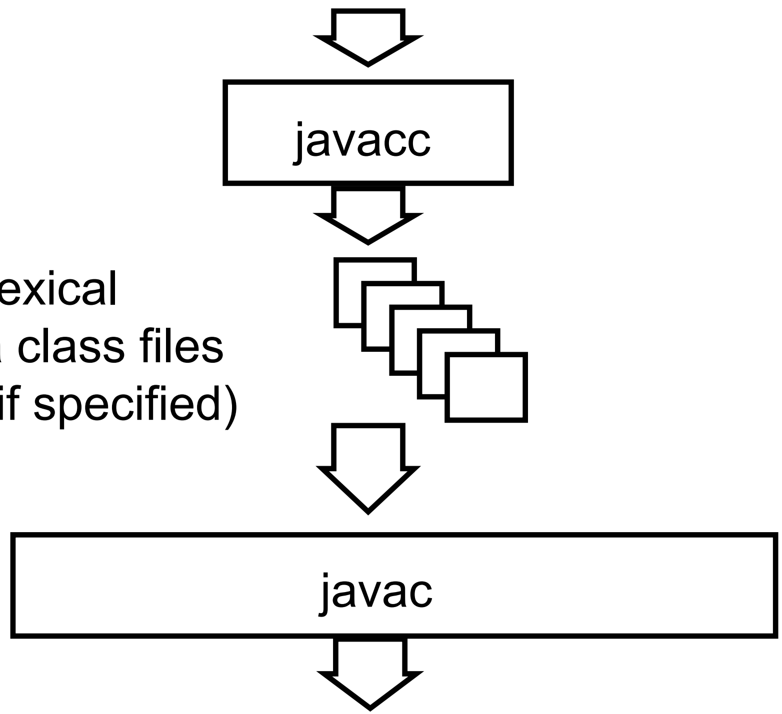
Example: simple expression evaluator

```
TOKEN :
{
  < NUM: (["0"-"9"])+ > | < EOL: "\n" >
}

int S() :
{ int s; }
{
    s=E() <EOL> { return s; }
  | <EOL>
  | <EOF>
}

int E() :
{ int e; int t; }
{
  e=T() (  "+" t=T() { e=e+t; }
             | "-" t=T() { e=e-t; } )*
    { return e; }
}
```

```
int T() :
{ int t; int f; }
{
  t=F() (  "*" f=F() { t=t*f; }
             | "/" f=F() { t=t/f; } )*
    { return t; }
}

int F() :
{ Token t; int result; }
{
    t=<NUM>
         { return Integer.parseInt(t.image); }
  | "(" result=E() ")"
         { return result; }
}
```

JavaCC

JavaCC specification
(includes both token specifications ***and***
grammar, possibly with actions)



Parsing and lexical
analysis Java class files
(with actions if specified)

combined token matcher and parser (executing actions if specified)

JavaCC actions

non-terminals can deliver values

we can declare some variables to use in actions

we can assign to variables from terminals and non-terminals

```
int E() :
{ int e; int t; }
{
  e=T() (  "+" t=T() { e=e+t; }
          | "-" t=T() { e=e-t; } )*
   { return e; }
}
```

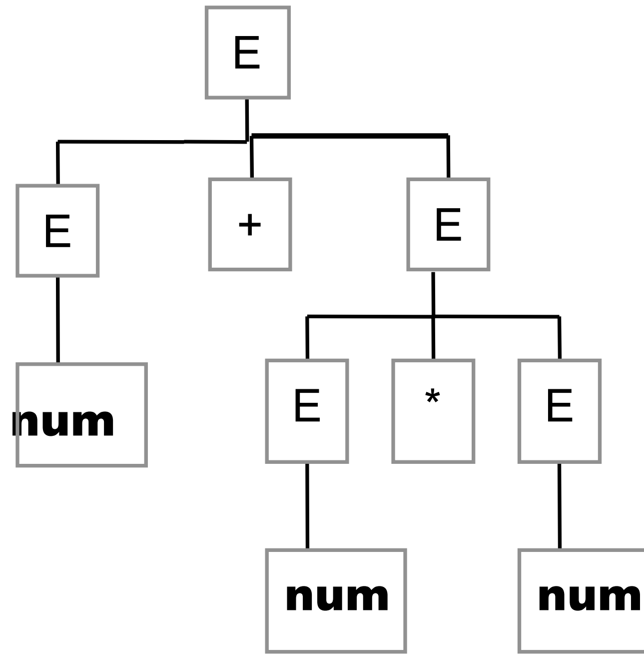
we can write any Java code in actions

this is where the non-terminal value is delivered

Abstract syntax trees

Abstract syntax for expressions

E ⟶ E * E | E / E | E **+** E | E **-** E | **num**



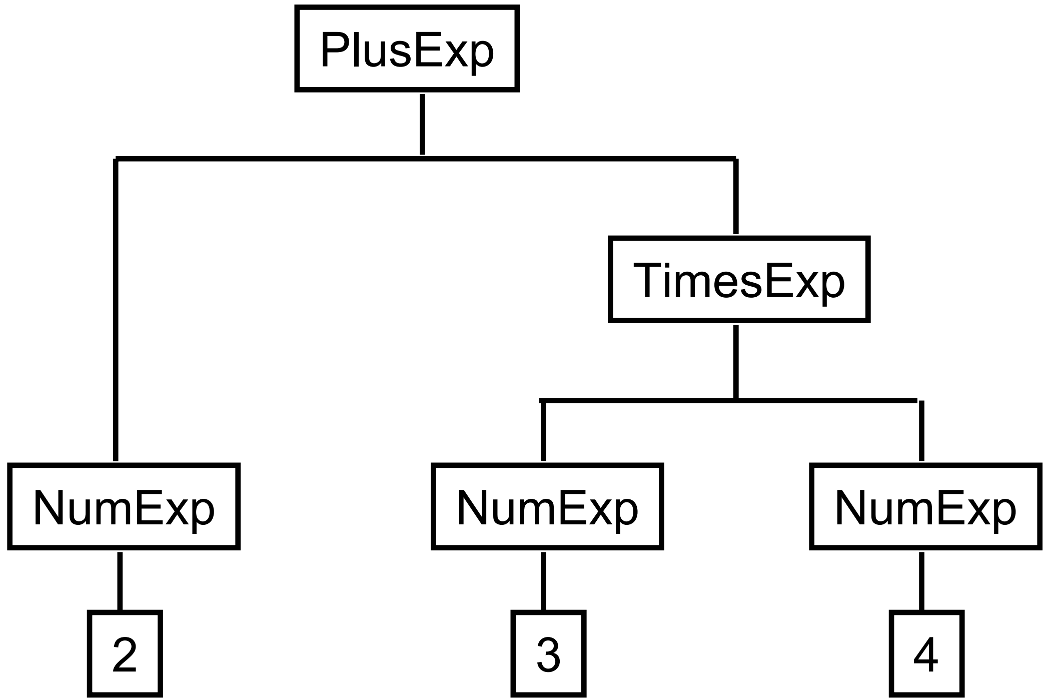
num+num*num

```
package syntaxtree;

public abstract class Exp {}

public class NumExp  extends Exp {
  private String f0;
  public NumExp (String n0) { f0=n0; }
}

public class PlusExp extends Exp {
  private Exp e1, e2;
  public PlusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
}
```

Abstract syntax tree representation

2+3*4

PlusExp(NumExp(2),TimesExp(NumExp(3),NumExp(4)))



Actions to create abstract syntax trees

```
Exp S() :
{ Exp s; }
{
    s=E() <EOL> { return s; }
  | <EOL>
  | <EOF>
}
```

```
Exp T() :
{ Exp t; Exp f; }
{
  t=F() (  "*" f=F() { t=new TimesExp(t,f); }
          | "/" f=F() { t=new DivideExp(t,f); } )*
    { return t; }
}
```

```
Exp E() :
{ Exp e; Exp t; }
{
  e=T() (  "+" t=T() { e=new PlusExp(e,t); }
          | "-" t=T() { e=new MinusExp(e,t); } )*
    { return e; }
}
```

```
Exp F() :
{ Token t; Exp result; }
{
    t=<NUM> { return new
                     NumExp(t.image); }
  |
     "(" result=E() ")" { return result; }
}
```

Using the abstract syntax tree

```
package syntaxtree;

public abstract class Exp {
  public abstract int eval();
}

public class NumExp  extends Exp {
  private String f0;
  public NumExp (String n0) { f0=n0; }
  public int eval() {
    return Integer.parseInt(f0);
  }
}
```

```
public class PlusExp extends Exp {
  private Exp e1, e2;
  public PlusExp(Exp a1, Exp a2) {
       e1=a1; e2=a2;
  }
  public int eval() {
       return e1.eval()+e2.eval();
  }
}
```

Main.java

```
root = parser.S();
System.out.println("Answer is "+root.eval());
```

JavaCC parsers and actions

- normally, the JavaCC grammar has semantic actions and values that are suited to creating the abstract syntax tree
  - the parser returns the root of the abstract tree when the parse completes successfully (here, S() returns a reference to the root object which is of class Exp)
- with the expression language, we simply wrote an eval method to calculate the value; this is not usual…
- instead, further methods are written that traverse the abstract tree to do useful things
  - typechecking
  - code generation
  - etc

Visitors – a better way to traverse the tree

- “Visitor pattern”
  - Visitor implements an interpretation.
  - Visitor object contains a visit method for each syntax-tree class
  - Syntax-tree classes contain “accept” methods
  - Visitor calls “accept”(what is your class?). Then “accept” calls the “visit” of the visitor
- allows us to create new operations to be performed by tree traversal *without* changing the tree classes
- visitors describe both:
  - actions to be performed at tree nodes, *and*
  - access to subtree objects from this node

Tree classes with accept methods for visitors

```
package syntaxtree;

import visitor.*;

public abstract class Exp {
  public abstract int accept(Visitor v);
}

public class NumExp  extends Exp {
  public String f0;
  public NumExp (String n0) { f0=n0; }
  public int accept(Visitor v) {
    return v.visit(this);
  }
}
```

```
public class PlusExp extends Exp {
  public Exp e1, e2;
  public PlusExp(Exp a1, Exp a2) { e1=a1;
e2=a2; }
  public int accept(Visitor v) {
    return v.visit(this);
  }
}
```

A calculator visitor

```
package visitor;
import syntaxtree.*;

public interface Visitor {
  public int visit(PlusExp n);
  public int visit(MinusExp n);
  public int visit(TimesExp n);
  public int visit(DivideExp n);
  public int visit(NumExp n);
}
```

Main.java

```
root = parser.S();
System.out.println("Answer is”
        +root.accept(new Calc()));
```

```
package visitor;
import syntaxtree.*;

public class Calc implements Visitor {
  public int visit (PlusExp n) {
    return n.e1.accept(this)+n.e2.accept(this);
  }
  public int visit (MinusExp n) {
    return n.e1.accept(this)-n.e2.accept(this);
  }
  public int visit (TimesExp n) {
    return n.e1.accept(this)*n.e2.accept(this);
  }
  public int visit (DivideExp n) {
    return n.e1.accept(this)/n.e2.accept(this);
  }
  public int visit (NumExp n) {
    return Integer.parseInt(n.f0);
  }
}
```

What you should do now…

- read and digest chapter 4
- look at MiniJava JavaCC definition for examples of lookahead
- understand visitors
- get ready to modify JavaCC specifications, and abstract syntax tree definitions, for coursework