City University London

IN2009
Language Processors

Weeks 4 & 5

Parsing II (abstract syntax)

Christian Cooper

Session Plan

Session 4: Parsing (abstract syntax)

Covered in weeks 4 and 5.

– MiniJava introduction and parsing
– Lookahead
– JavaCC grammars and semantic actions and values
– Simple expression evaluator
– Abstract syntax trees

Session Plan

– Using semantic actions to build abstract syntax trees
– Interpreting the trees
– Visitors
– MiniJava abstract syntax trees in Java
– JavaCC for generating MiniJava abstract syntax trees

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* * }
ClassDecl → **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* *
FormalList →

FormalRest → **,** *Type id*

Type → **int []**
Type → **boolean**
Type → **int**
Type → *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* *
ExpList →

ExpRest → **,** *Exp*

MiniJava Grammar IV

Exp → *Exp* *op* *Exp* `&& < + - *`
Exp → *Exp* **[** *Exp* **]**
Exp → *Exp* . **length**
Exp → *Exp* . *Id* **(** *ExpList* **)**
Exp → *INTEGER_LITERAL*
Exp → **true**
Exp → **false**
Exp → *id* *ambiguous?*
Exp → **this**
Exp → **new int [** *Exp* **]**
Exp → **new** *id* **( )**
Exp → **!** *Exp*
Exp → **(** *Exp* **)**

MiniJava JavaCC 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* * }
Statement → **if** (*Exp*) *Statement* **else** *Statement*
Statement → **while** (*Exp*) *Statement*
Statement → **System.out.println** (*Exp*) ;
Statement → *id* **=** *Exp* **;**
Statement → *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* **]**
Exp → *Exp* . **length**
Exp → *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).

Semantic actions

- 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 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

we can write any Java code in actions

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

this is where the non-terminal value is delivered

Abstract syntax trees

Abstract syntax for expressions

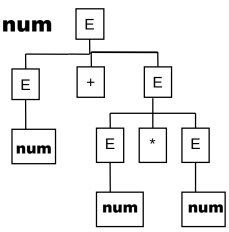
E ⟶ E ***** E **|** E / E **|** E **+** E **|** E **-** E **| num**

```
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; }
}
```

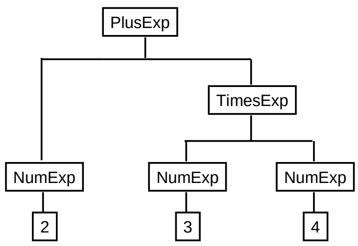


num+num*num

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)

JavaCC parsers and actions

- 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

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

Visitors

- Allow 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);
  }
}
```


Abstract Syntax for MiniJava

```
package syntaxtree;

Program(MainClass m, ClassDecList c1)
MainClass(Identifier i1, Identifier i2, Statement s)

abstract class ClassDecl
ClassDeclSimple(Identifier i, VarDeclList vl,
                 methodDeclList m1)
ClassDeclExtends(Identifier i, Identifier j,
                   VarDecList vl, MethodDeclList ml)

VarDecl(Type t, Identifier i)
MethodDecl(Type t, Identifier I, FormalList fl,
           VariableDeclList vl, StatementList sl, Exp e)
Formal(Type t, Identifier i)
```


Abstract Syntax for MiniJava

```
abstract class Type
IntArrayType()
BooleanType()
IntegerType()
IndentifierType(String s)

abstract class Statement
Block(StatementList sl)
If(Exp e, Statement s1, Statement s2)
While(Exp e, Statement s)
Print(Exp e)
Assign(Identifier i, Exp e)
ArrayAssign(Identifier i, Exp e1, Exp e2)
```


Abstract Syntax for MiniJava

```
abstract class Exp
And(Exp e1, Exp e2)          LessThan(Exp e1, Exp e2)
Plus(Exp e1, Exp e2)         Minus(Exp e1, Exp e2)
Times(Exp e1, Exp e2)        Not(Exp e)
ArrayLookup(Exp e1, Exp e2)  ArrayLength(Exp e)
Call(Exp e, Identifier i, ExpList el)
IntergerLiteral(int i)
True()                       False()
IdentifierExp(String s)
This()
NewArray(Exp e)              NewObject(Identifier i)

Identifier(String s) holds identifiers

--list classes:
ClassDecList() ExpList() FormalList() MethodDeclList()
StatementLIst() VarDeclList()
```


Syntax Tree Nodes - Details

```
package syntaxtree;
import visitor.Visitor;

public class Program {
  public MainClass m;
  public ClassDeclList cl;

  public Program(MainClass am, ClassDeclList
   acl) {
    m=am; cl=acl;
  }

  public void accept(Visitor v) {
    v.visit(this);
  }
}
```

StatementList.java

(all lists are like this)

```
package syntaxtree;
import java.util.Vector;

public class StatementList {
   private Vector list;
   public StatementList() {
      list = new Vector();
   }
   public void addElement(Statement n) {
      list.addElement(n);
   }
   public Statement elementAt(int i)  {
      return (Statement)list.elementAt(i);
   }
   public int size() {
      return list.size();
   }
}
```


Building AST lists in JavaCC

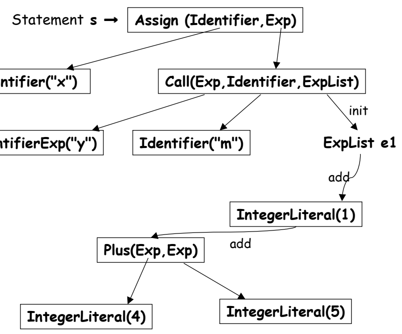
```
ExpList ExpressionList() :
{ Exp e1,e2;                                      init
  ExpList el = new ExpList();
}
{                                                 add
  e1=Expression()          { el.addElement(e1); }
  (  e2=ExpressionRest()   { el.addElement(e2); } )*
  { return el; }                                  add
}

Exp ExpressionRest() :
{ Exp e; }
{
  "," e=Expression()
  { return e; }
}
```


x = y.m(1,4+5)

Statement → AssignmentStatement
AssignmentStatement → Identfier$_1$ "=" Expression
Identifier$_1$ → <IDENTIFIER>
Expression → Expression$_1$ "." Identifier$_2$ "(" (ExpList)? ")"
Expression$_1$ → IdentifierExp
IdentifierExp → <IDENTIFIER>
Identifier$_2$ → <IDENTIFIER>
ExpList → Expression$_2$ ("," Expression$_3$)*
Expression$_2$ → <INTEGER_LITERAL>
Expression$_3$ → PlusExp → Expression "+" Expression
→ <INTEGER_LITERAL> "+" <INTEGER_LITERAL>

AST



MiniJava : Grammar & JavaCC

Program → *MainClass ClassDecl **

Program(MainClass, ClassDeclList)

```
Program Goal() :
{ MainClass m;
  ClassDeclList cl = new ClassDeclList();
  ClassDecl c;
}
{ m = MainClass() (c = ClassDecl()
 {cl.addElement(c);})*
  <EOF> {return new Program(m,cl); }
}
```


MiniJava : Grammar

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

ClassDecl → **class** *id* { *VarDecl * MethodDecl ** }
→ **class** *id* **extends** *id* { *VarDecl* MethodDecl ** }
ClassDeclSimple(…), ClassDecExtends(…)

VarDecl → *Type id ;*
VarDecl(Type, Identifier)

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

MethodDecl(Type,Identifier,FormalList,VarDeclList,Statement List,Exp)

MiniJava : Grammar

FormalList → *Type id FormalRest* *
→

FormalList() :- Formal(type,id), …

FormalRest → , *Type id*

Formal()

Type → **int []**
→ **boolean**
→ **int**
→ *id*

Type(), ArrayType(), BooleanType(), IntegerType(), IdentifierType()

MiniJava : Grammar

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

Statement(), Block(), If(), While(), Print(), Assign (), ArrayAssign()

ExpList → *Exp ExpRest* *
→

ExpRest → **,** *Exp*

MiniJava : Grammar

Exp → *Exp op Exp* **&& < + - ***
→ *Exp* **[** *Exp* **]**
→ *Exp* . **length**
→ *Exp* . *Id* **(** *ExpList* **)**
→ *INTEGER_LITERAL*
→ **true**
→ **false**
→ *id*
→ **this**
→ **new int [** *Exp* **]**
→ **new** *id* **( )**
→ **!** *Exp*
→ **(** *Exp* **)**

MainClass,ClassDecl in JavaCC

```
MainClass MainClass() :
{ Identifier i1,i2;
  Statement s; }
{
  "class" i1=Identifier() "{"
    "public" "static" "void" "main" "(" "String" "[" "]"
       i2=Identifier() ")" "{" s=Statement() "}" "}"
  { return new MainClass(i1,i2,s); }
}

ClassDecl ClassDeclaration() :
{ ClassDecl c; }
{
  (  LOOKAHEAD(3)
     c=ClassDeclarationSimple()
   | c=ClassDeclarationExtends()
   )
  { return c; }
}
```


FormalList, FormalRest in JavaCC

```
FormalList FormalParameterList() :
{ FormalList fl = new FormalList();   Formal f;
}
{  f=FormalParameter() { fl.addElement(f); }
  ( f=FormalParameterRest() { fl.addElement(f); } )*
  { return fl; }
}
Formal FormalParameter() :
{ Type t; Identifier i;
}
{  t=Type() i=Identifier()
  { return new Formal(t,i); }
}
Formal FormalParameterRest() :
{ Formal f; }
{  "," f=FormalParameter()
  { return f; }
}
```

FormalList → *Type id FormalRest* *
→

FormalRest → , *Type id*

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.
- Practice RegExps!

What you should do now…

- Read and understand about MiniJava and its abstract syntax trees and visitors
- (By the start of week 7) Begin the ***second*** coursework.

Schedule

- *This session continued*
- Monday 18th February, 2008
  - 11:00 - 12:50
  - C.348

Week 6 is a reading week!

- ***Semantic Analysis***
- Monday 3rd March, 2008
  - 11:00 - 12:50
  - C.348