City University London

IN2009

Language Processors

Session 5

Parsing II (abstract syntax)

Igor Siveroni

Session Plan

Session 4: Parsing (abstract syntax)
- Abstract syntax trees
- Using semantic actions to build abstract syntax trees
- Interpreting the trees
- Visitors

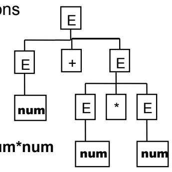
Abstract Syntax

Concrete Syntax: The grammar used for parsing. It's associated tree - Parse Tree - has a node per token. Inconvenient for later phases of the compiler - too much information!!

Abstract Syntax: Grammar used as interface between parsing and later phases of the compiler. The generated **Abstract Syntax Tree** (**AST**) eliminates redundant information e.g. punctuation tokens, but contains enough structure to drive semantic processing or even regenerate the input.

Example - Abstract syntax for expressions

E → E ***** E	TimesExp
E → E **/** E	DivExp
E → E **+** E	PlusExp
E → E **-** E	MinusExp
E → **num**	NumExp



num+num*num

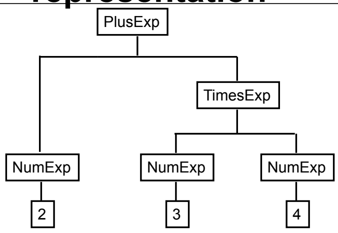
Abstract Syntax Trees

Abstract syntax for expressions represented by Java classes:

```
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; }
}
// similar for the other productions
```

Abstract syntax tree representation

AST of "2+3*4"



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

Can be generated with the following Java code:

```
new PlusExp(new NumExp(2),
        new TimesExp(new NumExp(3),new 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() { // Adition
      return e1.eval()+e2.eval();
  }
}

// similar for rest of classes
```

Main.java

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


Building AST lists with JavaCC

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

```
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)
```


StatementList.java

```
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();
   }
}
```

Or we could use a Java container

Building AST lists in JavaCC

```
StatementList StmList() :
{ Statement s1,s2;
  StatementListList sl = new StatementList();          init
}
{
   "{" s1=Statement()        { sl.addElement(s1); }    add
   (   s2=Statement()        { sl.addElement(s2); }  )*  "}"     add
  { return sl; }
}

Statement Statement() :
{ Statement s; }
{
  s = StmList() | s = IfStm() | …
  { return s; }
}
```


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 (for example, 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 was enough for our example but…
- In a compiler, further methods are written that traverse the abstract tree to do useful things
  – typechecking
  – code generation, etc
- Implementation of such functionality outside the abstract syntax classes improves modularity.
- Technique used: The Visitor Pattern

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);
  }
}
```


Tasks Reading Week

- Recommended reading: Chapters 1-4 from Modern Compiler Implementation book.
- Finish Lab 5 (out next Monday). You should:
  - Understand visitors
  - Be able to create/modify JavaCC specifications and abstract syntax tree definitions.
- Complete Test 1. (out next week)