Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue&Thu 12:00-13:15 @SH 212, Lab: Fri 10:00-11:15/11:30-12:45 @EB 103

Lab 6: AST Generation

Learning Objectives Upon successful completion, students will be able to:

- use the provided AST representation to build arbitrary ASTs;
- insert semantic actions into a JavaCC parser program to generate an AST.

Preparation

Download from D2L the file `lab6.zip` and unzip it. Enter the directory `lab6`.

AST Representation

For this lab, we are using the same AST representation that will be used in Assignment 3 and other projects of this course. In this representation, AST nodes are defined as Java classes. For the convenience of a single location, they are defined as *static nested classes* inside a wrapper class called `Ast`. A reference to any AST node constructor or any method needs to include the prefix `Ast.`, *e.g.* `Ast.Program()`, `Ast.Id()`.

A quick listing of all available nodes in this AST representation is shown below:

- *Declaration Nodes* — `Program`, `ClassDecl`, `MethodDecl`, `VarDecl`, and `Param`.
- *Type Nodes* — `IntType`, `BoolType`, `ArrayType`, and `ObjType`.
- *Statement Nodes* — `Block`, `Assign`, `CallStmt`, `If`, `While`, `Print`, and `Return`.
- *Expression Nodes* — `Binop`, `Unop`, `Call`, `NewArray`, `ArrayElm`, `NewObj`, `Field`, `Id`, `This`, `IntLit`, `BoolLit`, and `StrLit`.

The full specification of the AST classes is in the program file, `ast/Ast.java`. To access these classes in a program, include the package `ast`:

```
import ast.*;
```

The AST representation of a program can be printed out for external viewing:

```
Ast.Program p = new Ast.Program(new ArrayList<Ast.ClassDecl>());
System.out.print(p);
```

HelloAst.java — Manual Construction of an AST

This demo program shows how to manually build an AST for a miniJava "hello-world" program. The source program and the target AST's print out are shown below:

"Source Program: Hello.java"

```
// miniJava
class Hello {
  public static void main(String[] a) {
    System.out.println("Hello World!");
  }
}
```

```
# AST Program
ClassDecl Hello
  MethodDecl void main ()
   Print "Hello World!"
```

Look inside the program `HelloAst.java` to see how the AST is constructed:

"from HelloAst.java"

```
class HelloAst {
  public static void main(String [] args) {
    // Prepare arguments for constructing a MethodDecl
    List<Ast.Param> pl = new ArrayList<Ast.Param>();     // empty param list
    List<Ast.VarDecl> vl = new ArrayList<Ast.VarDecl>(); // empty var list
    List<Ast.Stmt> sl = new ArrayList<Ast.Stmt>();       // empty stmt list

    // Construct a print stmt and add it to stmt list
    Ast.Exp arg = new Ast.StrLit("Hello World!");       // create an arg
    Ast.Stmt s = new Ast.Print(arg);                    // create a print stmt
    sl.add(s);                                          // add stmt to list

    // Construct a MethodDecl ---
    //   Ast.MethodDecl(Ast.Type rt, String m, List<Ast.Param> pl,
    //                  List<Ast.VarDecl> vl, List<Ast.Stmt> sl)
    Ast.MethodDecl md = new Ast.MethodDecl(null,        // null represents 'void'
                                           "main",      // method's name
                                           pl,          // formal params
                                           vl,          // local var decls
                                           sl);         // method body
    List<Ast.MethodDecl> ml = new ArrayList<Ast.MethodDecl>();
    ml.add(md);

    // Construct a ClassDecl ---
    //   Ast.ClassDecl(String nm, String pnm,
    //                 List<Ast.VarDecl> vl, List<Ast.MethodDecl> ml)
    Ast.ClassDecl cd = new Ast.ClassDecl("Hello",       // class name
                                         null,          // parent's name
                                         vl,            // field decls
                                         ml);           // method decls
    List<Ast.ClassDecl> cl = new ArrayList<Ast.ClassDecl>();
    cl.add(cd);

    // Construct a Program ---
    //   Ast.Program(List<Ast.ClassDecl> cl)
    Ast.Program p = new Ast.Program(cl);        // create the whole AST
    System.out.print(p);                        // dump out the AST
  }
}
```

Exercise: Following the above example, construct an AST for the source program, `Sum.java`, which computes the sum of three variables (see below). Call your AST-construction program `SumAst.java`. Copy and use any code you think is useful from `HelloAst.java`. The source program and the expected output are shown below:

"Source Program: Sum.java"

```
class Sum {
  public static void main(String[] ignore) {
    int x = 1;
    int y = 2;
    int z = 3;
    int sum;
    sum = x + y + z;
    System.out.println(sum);
  }
}
```

```
# AST Program
ClassDecl Sum
  MethodDecl void main ()
   VarDecl IntType x 1
   VarDecl IntType y 2
   VarDecl IntType z 3
   VarDecl IntType sum ()
   Assign sum (Binop + (Binop + x y) z)
   Print sum
```

Stmt1.jj — AST Generation for a Simple Assignment Statement

Consider the grammar for a simplified assignment statement:

$S \rightarrow$ id = intlit ;

AST generation for this statement is straightforward — just construct an `Ast.Assign` node:

"from Stmt1.jj"

```
// S -> <ID> "=" <IntLit> ";"
//
Ast.Stmt S():
{ Token tkn1, tkn2; Ast.Id id; Ast.IntLit intlit; }
{
  tkn1=<ID> "=" tkn2=<IntLit> ";"
  { id = new Ast.Id(tkn1.image);
    intlit = new Ast.IntLit(Integer.parseInt(tkn2.image));
    return new Ast.Assign(id, intlit); }
}
```

Exercises:

1. Compile and test this program. (You may use `test1.in` as input.)
2. Add a print statement to this program:

   $S \rightarrow$ "print" intlit ;

   Compile and test it. (You may use `test2.in` as input.)

Stmt2.jj — AST Generation for a Simple Call Statement

Consider the grammar for a simple call statement:

$S \rightarrow$ id (id) ;

AST generation is still straightforward — construct an `Ast.CallStmt` node:

"from Stmt2.jj"

```
// S -> Id "(" Id ")" ";"
//
Ast.Stmt S():
{ List<Ast.Exp> args = new ArrayList<Ast.Exp>();
  Ast.Id id1, id2; }
{
  id1=Id() "(" id2=Id() ")" ";"
  { args.add(id2);
    return new Ast.CallStmt(new Ast.This(), id1.nm, args); }
}

// Id -> <ID>
//
Ast.Id Id():
{ Token tkn; }
{
  tkn=<ID> { return new Ast.Id(tkn.image); }
}
```

Notice the creation of an `Ast.This` object. In Java, a non-static method call needs to be invoked off an object. If the object is not explicitly specified, the current object, `this`, is used.

Exercise: Now, putting the assignment and call statement together, we have the following grammar:

$S \rightarrow$ id = intlit ;
$S \rightarrow$ id (id) ;

This is not an LL(1) grammar, since the two productions share a common prefix. But it can be easily fixed with left-factoring:

$S \rightarrow$ id (= intlit | (id)) ;

Now encode this grammar in a JavaCC program and add semantic actions to generate an AST. Compile and test your new program. (You may use both `test1.in` and `test3.in` as input.)

Stmt3.jj — AST Generation for a More Complex Call Statement

Consider a more complex call statement:

$S \rightarrow$ id [. id] (id) ; ([] denotes the EBNF optional operator)

AST generation is a little more complex:

"from Stmt3.jj"

```
// S -> <ID> ["." <ID>] "(" <ID> ")" ";"
//
Ast.Stmt S():
{ List<Ast.Exp> args = new ArrayList<Ast.Exp>();
  Ast.Id id1, id2=null, id3; Ast.Stmt s; }
{
  id1=Id() [ "." id2=Id() ] "(" id3=Id() ")" ";"
  { args.add(id3);
    if (id2 == null)
      return new Ast.CallStmt(new Ast.This(), id1.nm, args);
    else
      return new Ast.CallStmt(id1, id2.nm, args);
  }
}
```

Exercises:

1. Generate AST for a more complex assignment statement:

   $S \rightarrow$ id [. id] = intlit ;

   Note: If both ids get matched, then the left-hand-side of the assignment should be represented by an `Ast.Field` node.

2. (*) Combine the two complex forms of assignment and call statement into a single grammar, and generate AST for it.

Expr1.jj – AST Generation for a Simple Expression

Now consider the following simple expression grammar:

$E \rightarrow$ id { op id }
op $\rightarrow + \mid -$

({ } and | are EBNF operators)

AST generation for this grammar:

"from Expr1.jj"

```
// E -> Id {Op Id}
//
Ast.Exp E():
{ Ast.Id id; Ast.BOP op; Ast.Exp e; }
{
  id=Id() { e = id; }
  ( op=Op() id=Id() { e = new Ast.Binop(op,e,id); } )*
  { return e; }
}

// Op -> "+"|"-"
//
Ast.BOP Op(): {}
{
  "+" { return Ast.BOP.ADD; }
| "-" { return Ast.BOP.SUB; }
}
```

If there is just a single ID, the return AST is just an `Ast.Id` object. Otherwise, recursion starts, and every time a new ID is accepted, a new `Ast.Binop` object is created, combining the subexpression built so far with the new ID.

Exercises:

1. Compile and test the program. On input:

   ```
   a + b - c - d
   ```

   you should see output:

   ```
   (Binop - (Binop - (Binop + a b) c) d)
   ```

   Notice that the AST shows left-associativity.

2. Now consider the following grammar:

   $E \rightarrow$ id [op E]
   op $\rightarrow + \mid -$

   Encode this grammar to JavaCC and insert semantic actions to generate an AST.

3. Compile and test the new program. With the same input, `a + b - c - d`, do you see left-associativity or right-associativity in the output?

4. Define an attribute grammar for the generation of AST for this grammar.

Expr2.jj — AST Generation for Another Simple Expression

Consider the grammar from the exercises above:

$E \rightarrow \mathsf{id}\ [\ \mathsf{op}\ E\ ]$
$\mathsf{op} \rightarrow + \mid -$

Look at the following set of semantic actions:

"from Expr2.jj"

```
// E -> Id [Op E]
//
Ast.Exp E(Ast.Exp e0, Ast.BOP op0):
{ Ast.Id id; Ast.BOP op; Ast.Exp e,e2; }
{
  id=Id() { e = (e0==null)? id : new Ast.Binop(op0, e0, id); }
  [ op=Op() e=E(e,op) ]
  { return e; }
}
```

If you compile and run this program, on input:

```
a + b - c - d
```

you'll see output:

```
(Binop - (Binop - (Binop + a b) c) d)
```

In other words, while the grammar is right-recursive, the generated AST demonstrates left-associativity.

Analysis: In this version, instead of calling the parsing routine `E()` recursively and use the result as the right operand in the current AST node, we pass the current node's information (the left operand and the operator) down through arguments to the parsing routine `E()`. In the recursive call, the incoming components will be combined with the first recursive item into a single node (and passed further down into the recursion), resulting in the desired left-associativity.

This example illustrates the use of inherited attributes, instead of the more natural synthesized attributes.

Exercise: (*) Try to define an attribute grammar for the generation of AST for this grammar. (*Hint:* You may need to define multiple attributes, some inherited, some synthesized.)