MINI (v1.6) Abstract Syntax Tree

Jingke Li

Portland State University

AST Generation

- *Program Organization:*

  ast/ — containing AST node definitions
  parser/ — containing the parser files
  tst/ — containing the test files
  runp — a script for running a regression test over a set of test files

- *AST Nodes:*

  – All AST nodes are derived from a single `Ast` class
  – There are four groups of AST nodes: types, statements, expressions, and lists

- *Modification based on Parser0:*

  – Each parsing routine needs to return a proper AST node
  – Some routines may need to take input parameter(s)

The Top-Level AST Node

```
package ast;

public abstract class Ast {
  public static void DUMP(String s) { System.out.print(s); }

  public static void DUMP(String name, List l) {
    System.out.print("(" + name + " ");
    if (l!=null) l.dump();
    System.out.print(") ");
  }
  public static void DUMP(Type t) { ... };
  public static void DUMP(Stmt s) { ... };
  public static void DUMP(Exp e)  { ... };
  public abstract void dump();

  public abstract void accept(VoidVI v);
}
```

AST Nodes — Types

```
public abstract class Type extends Ast { ... }

public class BasicType extends Type {
  public static final int Int=0, Float=1, Bool=2;
  public int typ;

  public BasicType(int t) { typ=t; }
  public String typName(int typ) { ... }
  public String toString() { return typName(typ); }
}
public class ArrayType extends Type {
  public Type et;
  ...
}
public class ObjType extends Type {
  public Id cid;
  ...
}
```

AST Nodes — Statements

```
public abstract class Stmt extends Ast { ... }

public class Assign extends Stmt {
  public Exp lhs, rhs;
  public Assign(Exp e1, Exp e2) { lhs=e1; rhs=e2; }
  public void dump() {
    DUMP("\n (Assign "); DUMP(lhs); DUMP(rhs); DUMP(") ");
  }
}

public class If extends Stmt {
  public Exp e;
  public Stmt s1, s2;
  public If(Exp ae, Stmt as1, Stmt as2) { e=ae; s1=as1; s2=as2; }
  public void dump() {
    DUMP("\n (If "); DUMP(e); DUMP(s1); DUMP(s2); DUMP(") ");
  }
}

...
```

AST Nodes — Expressions

```
public abstract class Exp extends Ast { ... }

public class Binop extends Exp {
  public static final int ADD=0, SUB=1, MUL=2, DIV=3, AND=4, OR=5;
  public int op;
  public Exp e1, e2;
  public Binop(int o, Exp ae1, Exp ae2) { op=o; e1=ae1; e2=ae2; }
  private void dumpOp(int op) { ... }
  public void dump() { ... }
}
public class ArrayElm extends Exp {
  public Exp array, idx;
  ...
}
public class Id extends Exp {
  public String s;
  ...
}
```

AST Nodes — Lists

```
public class AstList extends Ast {
  private Vector<Ast> list;
  public AstList() { list = new Vector<Ast>(); }
  public void add(Ast n) { list.addElement(n); }
  public void addAll(AstList l) { list.addAll(l.list); }
  public Ast elementAt(int i) { return list.elementAt(i); }
  public int size() { return list.size(); }
  public void dump() { ... }
}

public class ClassDeclList extends AstList {
  public ClassDeclList() { super(); }
  public void add(ClassDecl n) { super.add(n); }
  public ClassDecl elementAt(int i)  {
    return (ClassDecl)super.elementAt(i);
  }
}

...
```

Generating AST with JavaCC

Create a proper return value for each parsing routine:

```
Program Program() :
ClassDecl ClassDecl() :
MethodDecl MethodDecl() :
...
Type Type() :
Type BasicType() :
...
Stmt Statement() :
Stmt Block() :
Stmt AssignOrCall() :
Stmt If() :
Stmt While() :
Stmt Print() :
Stmt Return() :
```

```
Exp InitExpr() :
Exp Expr() :
Exp OrTail(Exp e0) :
Exp AndExpr() :
Exp AndTail(Exp e0) :
Exp ArithExpr() :
Exp ArithTail(Exp e0) :
Exp Term() :
Exp TermTail(Exp e0) :
Exp Factor() :
Exp Lvalue() :
...
Id Id() :
Exp Intval() :
```

Declaration Routine Examples

```
// Program -> ClassDecl {ClassDecl} <EOF>
Program Program() :
{ ClassDecl c; ClassDeclList cl = new ClassDeclList(); }
{ c=ClassDecl() { cl.add(c); } (c=ClassDecl() { cl.add(c); })* <EOF>
  { return new Program(cl); }
}

// ClassDecl -> "class" <ID> ["extends" <ID>]
//                  '{' {VarDecl} {MethodDecl} '}'
ClassDecl ClassDecl() :
{ Id cid, pid = null; VarDecl v; MethodDecl m;
  VarDeclList vl = new VarDeclList();
  MethodDeclList ml = new MethodDeclList(); }
{ "class" cid=Id() ["extends" pid=Id()]
    "{" (v=VarDecl() { vl.add(v); })*
        (m=MethodDecl() { ml.add(m); })* "}"
  { return new ClassDecl(cid,pid,vl,ml); }
}
```

Statement Routine Examples

```
// If -> "if" '(' Expr ')' Statement ["else" Statement]
Stmt If():
{ Exp e; Stmt s1, s2=null; }
{
  "if" "(" e=Expr() ")" s1=Statement()
    [LOOKAHEAD(1) "else" s2=Statement()]
  { return new If(e,s1,s2); }
}

// Print -> "System.out.println" '(' [Expr|<STRVAL>] ')' ';'
Stmt Print():
{ Exp e=null; }
{
  "System.out.println" "(" [ e=Expr() | e=Strval() ] ")" ";"
  { return new Print(e); }
}
```

Expression Routine Examples

Use parameter to carry value into a parsing routine when needed:

```
// Expr -> AndExpr OrTail
Exp Expr() :
{ Exp e1, e2 = null; }
{ e1=AndExpr() e2=OrTail(e1)
  { return e2==null ? e1 : e2; }
}

// OrTail -> ["||" AndExpr OrTail]
Exp OrTail(Exp e0) :
{ Exp e1 = null, e2 = null; }
{ ["||" e1=AndExpr() { e1 = new Binop(Binop.OR,e0,e1); }
    e2=OrTail(e1)]
  { return e2==null ? e1 : e2; }
}
```

Handling Lvalues

Lvalue represents a collection of syntax forms, e.g. id, array element, and object field (field):

```
Lvalue -> ["this" '.'] <ID> {'.' <ID>} ['[' Expr ']']
```

There is no AST node corresponding to Lvalue directly. The Lvalue parsing routine should return an AST node that correponds to the specific syntax form.

Examples:

Syntax form	*Should return*
`<ID>`	`Id()`
`"this" '.' <ID>`	`Field(This(), Id())`
`<ID> '[' Expr ']'`	`ArrayElm(Id(), Expr())`
`"This" '.' <ID> '[' Expr ']'`	`ArrayElm(Field(This(), Id()), Expr())`

Handling Lvalues (cont.)

```
Exp Lvalue(): { ... }
{
  [ <THIS> "." {...} ] id=Id() // construct either an Id node
                               // or a Field node
  ( ( "[" e=Expr() "]" // construct an ArrayElm node
    | "." id=Id()      // construct a Field node ) )*
  // return the constructed node
}
```

Example: The expression `f.g.a[5]` shoud be processed as follows

```
f.g      - Field(Id("f"), Id("g"))
f.g.a    - Field(Field(Id("f"), Id("g")), Id("a"))
f.g.a[5] - ArrayElm(Field(Field(Id("f"), Id("g")),
                    Id("a")), Intval(5))
```

Handling Calls

Method call has the following syntax:

```
Call -> Lvalue '(' [Params] ')'
```

To construct a Call node, we need obj (object), mid (method id), and args (arguments). The first two pieces have to be extracted out from the Ast node returned by the recursive call to Lvalue.

```
Exp Call() { ... }
{
  e=Lvalue() "(" [ el=Params() ] ")"
  { if (e instanceof Id)
      // return Call(This(), e, el)
    else if (e instanceof Field)
      // return Call(((Field) e).obj, ((Field) e).var, el)
    else
      // illegal case
  }
}
```