Symbol Table in MINI Compiler (v1.6)

Jingke Li

Portland State University

Symbol Table in MINI Compiler

Symbol Table Representation — Multi-table organization:

- `Table.java` — A top level table. Contains a table of class recs.
- `ClassRec.java` — For class decls. Contains a table of class-variable recs and a table of method recs.
- `MethodRec.java` — For method decls. Contains a table of parameter recs and a table of local-variable recs.
- `VarRec.java` — For storing variable/parameter information.

Code Organization — Visitor pattern over the AST hierarchy:

- `TypeVI.java` — the visitor interface to use
- `SymbolVisitor.java` — an implementation of `TyteVI`:
  - Collect info from (only) decl nodes
  - Keep track of current class and method scopes
  - Handle class hierarchy

Table.java

```
public class Table {
  private Hashtable<String,ClassRec> classes;

  public Table() { classes = new Hashtable<String,ClassRec>(); }

  public void addClass(Id cid) throws SymbolException {
    Object old_entry = classes.put(cid.s, new ClassRec(cid));
    if (old_entry != null)
      throw new SymbolException("Class " + cid.s + " already defined");
  }
  public ClassRec getClass(Id cid) throws SymbolException {
    ClassRec c = (ClassRec) classes.get(cid.s);
    if (c == null) throw new SymbolException("Class not defined");
    return c;
  }
  public Method getMethod(ClassRec c, Id mid) ...
  public Var getVar(ClassRec c, Method m, Id vid) ...
  public void show() ...
}
```

ClassRec.java

```
public class ClassRec {
  private Id id;
  private ClassRec parent;
  private Vector<VarRec> class_vars;
  private Hashtable<String,MethodRec> methods;

  public ClassRec(Id cid)  { ... }
  public VarRec getClassVar(Id vid) { ... }
  public VarRec getClassVarAt(int i) { ... }
  public void addClassVar(Id vid, Type type, Exp e) ...
  public MethodRec getMethod(Id mid) { ... }
  public void addMethod(Id cid, Id mid, Type rtype) ...
  public int varCnt()      { return class_vars.size(); }
  public void linkParent(ClassRec p) {
    parent = p;
    int start = getVarStartIdx();
    for (int i = 0; i < class_vars.size(); i++)
      ((VarRec)class_vars.elementAt(i)).setIdx(start + i + 1);
  }
}
```

MethodRec.java

```
public class MethodRec {
  private Id id;
  private Type rtype;
  private Vector<VarRec> params;
  private Vector<VarRec> locals;

  public MethodRec(Id id, Type type) { ... }
  public int paramCnt()    { return params.size(); }
  public int localCnt()    { return locals.size(); }
  public VarRec getParam(Id vid) { ... }
  public VarRec getParamAt(int i) { ... }
  public VarRec getLocal(Id vid) { ... }
  public VarRec getLocalAt(int i) { ... }
  public void addParam(Id vid, Type type) ...
  public void addLocal(Id vid, Type type) ...
}
```

VarRec.java

```
public class VarRec {
  public static final int CLASS=0, LOCAL=1, PARAM=2;
  private Id id;
  private Type type;
  private int kind;  // one of the three categories
  private int idx;   // position in its scope
  private Exp init;  // initial value

  public VarRec(Id vid, Type vtype, int vkind, int vidx) { ... }
  public VarRec(Id vid, Type vtype, int vkind, int vidx, Exp e) { ... }
  public void setIdx(int vidx) { idx = vidx; }
}
```

TypeVI.java — a Visitor Interface

```
public interface TypeVI {
  public void visit(Program n) throws Exception;
  public void visit(ClassDeclList n) throws Exception;
  ...
  public void visit(ClassDecl n) throws Exception;
  public void visit(MethodDecl n) throws Exception;
  ...
  public Type visit(IntType n);
  public Type visit(BoolType n);
  ...
  public void visit(Block n) throws Exception;
  public void visit(Assign n) throws Exception;
  ...
  public Type visit(Binop n) throws Exception;
  public Type visit(Relop n) throws Exception;
  ...
  public Type visit(Int n);
  public Type visit(Bool n);
  ...
}
```

SymbolVisitor.java

```
package symbol;
import ast.*;
import java.util.Hashtable;
import java.util.Vector;

public class SymbolVisitor implements TypeVI {
  public Table symTable;         // the top-scope symbol table
  private ClassRec currClass;    // the current class scope
  private MethodRec currMethod; // the current method scope
  private boolean hasMain;       // whether "main" method is defined

  public SymbolVisitor() { ... }

  public void visit(Program n) throws Exception {
    n.cl.accept(this);
    if (!hasMain)
      throw new SymbolException("Method main is missing");
    setupClassHierarchy(n.cl); // establish class hierarchy
  }
```

SymbolVisitor.java — Declarations

These are the routines you need to complete:

```
public void visit(ClassDecl n) throws Exception {
  // add a ClassRec to symTable
  // recursively process vl list and ml list
}
public void visit(MethodDecl n) throws Exception {
  // add a MethodRec to the current ClassRec
  // recursively process fl list and vl list
  // if the method is 'main', check for violations
  //   of main method's rules
}
public void visit(VarDecl n) throws Exception {
  // decide whether the var is a local var or a class var
  // (hint: use env variables currClass and currMethod)
  // add a VarRec to the proper ClassRec of MethodRec
}
public void visit(Formal n) throws Exception {
  // add a VarRec to the current MethodRec's param list
}
```

SymbolVisitor.java — Lists and Types

```
// LISTS --- use default traversal
public void visit(List n) throws Exception {}
public void visit(ClassDeclList n) throws Exception {
  for (int i = 0; i < n.size(); i++)
    n.elementAt(i).accept(this);
}
public void visit(VarDeclList n) throws Exception {
  for (int i = 0; i < n.size(); i++)
    n.elementAt(i).accept(this);
}
...
// TYPES --- return the nodes themselves
public Type visit(IntType n) { return n; }
public Type visit(BoolType n) { return n; }
public Type visit(ObjType n) throws Exception { return n; }
...
```

SymbolVisitor.java — Stmts and Exprs

Nothing to implement in these two groups:

```
  public void visit(StmtList n) throws Exception {}
  public void visit(Block n) throws Exception {}
  public void visit(Assign n) throws Exception {}
  ...

  public void visit(ExpList n) throws Exception {}
  public Type visit(Binop n) throws Exception { return null; }
  public Type visit(Relop n) throws Exception { return null; }
  ...
}
```

MINI's Scope Issues

In MINI, classes and methods can create new scopes.

- A method's scope is enclosed in its class's scope.
- A class's scope is enclosed in its parent's scope (if exists).

```
class A {
  int i;
}
class B extends A {
  public int foo() {
    return i;  // i is defined in a scope two levels up
  }
}
```

We could use two variables *currClass* and *currMethod* to keep track of the scope info during symbol processing — at the beginning of `ClassDecl` and *MethodDecl* visit routines, set these variables to point to their corresponding symbol-table recs; and at the end, reset them.

MINI's Scope Issues (cont.)

- Within a class scope, the data fields and the methods are in two *disjoint* sub-scopes. In other words, a class variable and a method can share the same name. E.g.

```
class A {
  int j;
  public int j() { return 3; } // OK
}
```

- Within a method scope, parameters and local variables *can not* share the same name.

```
public int foo(int i) {
  int i=5;    // Error!
  return i;
}
```

MINI's Recursive Definitions

In MINI, class definitions and method definitions can both be *mutually* recursive. E.g. the following examples are both OK:

```
class A {
  public void foo(int i) {
    B b = new B(1);
    zoo(1);
  }
  public void zoo(int j) {
    foo(2);
  }
}
class B {
  public void bar() {
    A a = new A(2);
  }
}
```

How to handle this feature?

MINI's Recursive Definitions (cont.)

Answer: Need to process class and method declarations *twice*:

- In the first visit (done by `SymbolVisitor`), information regarding classes (or methods) are recorded in the symbol table
- In the second visit (done later by `CheckerVisitor`), semantic correctness is checked

So if the definition of class `B` is missing in the previous example, the `SymbolVisitor` pass would not be able to detect the error. But the next `CheckerVisitor` (type-checking) pass will catch this error.

Setting up Class Hierarchy

```
class B extends A {
  int k;
}
class A {
  int i;
  int j;
  public int foo(int i)
    { int y; return i+this.i; }
}
```

```
Symbol Table:
Class A (pid=null):
  [cl var] (1) i int
  [cl var] (2) j int
 <method> A.foo (rtype=int):
  [param] (1) i int
  [local] (1) y int
Class B (pid=A):
  [cl var] (3) k int
```

- The inheritance relationship among classes need to be captured with the *parent* pointers in their ClassRecs.
- Furthermore, the start index for a subclass's variables need to take into consideration all ancestor classes' variables.

The difficulty is that a subclass may be defined before its parent class in the program.

Setting up Class Hierarchy (cont.)

Solution: Perform a topological sort on class decls based on their inheritance relationship, and process parent decl first.

```
private void setupClassHierarchy(ClassDeclList cl) throws Exception {
  Vector<ClassDecl> work = new Vector<ClassDecl>();
  Vector<String> done = new Vector<String>();
  for (int i=0; i<cl.size(); i++)
    work.add(cl.elementAt(i));
  while (work.size() > 0) {
    for (int i=0; i<work.size(); i++) {
      ClassDecl cd = (ClassDecl) work.elementAt(i);
      if (cd.pid != null) {
        if (!done.contains(cd.pid.s)) continue;
        ClassRec cr = symTable.getClass(cd.cid);
        ClassRec pr = symTable.getClass(cd.pid);
        cr.linkParent(pr);
      }
      done.add(cd.cid.s);
      work.remove(cd);
    }
  }
}
```