Session Plan

- Session 5: ***Semantic analysis***
  - symbol tables
    - » environments
    - » hash tables
    - » symbol table for MiniJava
  - typechecking

Semantic analysis (1)

- connects variable (and type/class, and function) definitions to their uses
- checks that each use matches an appropriate declaration
  - must also use scope rules of language…
- checks that each expression/part of the program is of correct type
- translates abstract syntax into simpler representation suitable for generating machine code

Semantic analysis (2)

- collect all identifiers in symbol table(s) with attributes such as: type; scope; for a function or method, parameter types & return type
  - by traversal of declaration ASTs
- check that applied occurrences of identifiers have related declarations (are in table) at right scope
  - by traversal of rest of program ASTs
- check that types of (sub-)expressions and parts of statements are of expected type
  - by traversal of rest of program ASTs and collecting types
  - eg actual param matches type of formal; in A+B, A,B are ints

Symbol tables (environments)

- map identifiers to their types & locations
- identifiers (variable names, function or method names, type or class names) are bound to "meanings" (bindings) in symbol tables
  - eg type of variable name
  - eg number and type of parameters of function
  - eg scope
- perform lookup in symbol table when there is a *use* (non-defining occurrence) of identifier

scope for each local variable in which it is visible:

```
class X { int y;… }
local variable in a Java
method?
```

discard identifier bindings local to scope at end of scope analysis

Environment is a set of bindings → e.g.

$$\sigma_0 = \{\texttt{g} \rightarrow \texttt{string}, \texttt{a} \rightarrow \texttt{int}\}$$

Example

```
1. class C {
2.   int a; int b; int c;
3.   public void m() {
4.     System.out.println(a+c);
5.     int j = a+b;
6.     String a = "hello"
7.     System.out.println(a);
8.     System.out.println(j);
9.     System.out.println(b);
10.  }
11.}
```

1	σ_0 is starting environment
2	$\sigma_1 = \sigma_0 + \{$`a`$\rightarrow$`int`,`b`$\rightarrow$`int`,`c`$\rightarrow$`int`$\}$
3	
4	lookup ids `a,c` in σ_1
5	$\sigma_2 = \sigma_1 + \{$`j`$\rightarrow$`int`$\}$
6	$\sigma_3 = \sigma_2 + \{$`a`$\rightarrow$`string`$\}$
7	lookup a in σ_3
8	lookup j in σ_3
9	lookup `b` in σ_1
10	discard σ_3 revert to σ_1
11	discard σ_1 revert to σ_0

Need to deal with clashes (different bindings for same symbol)

σ_2 : `a`$\rightarrow$`int` , σ_3 : `a`$\rightarrow$`string`

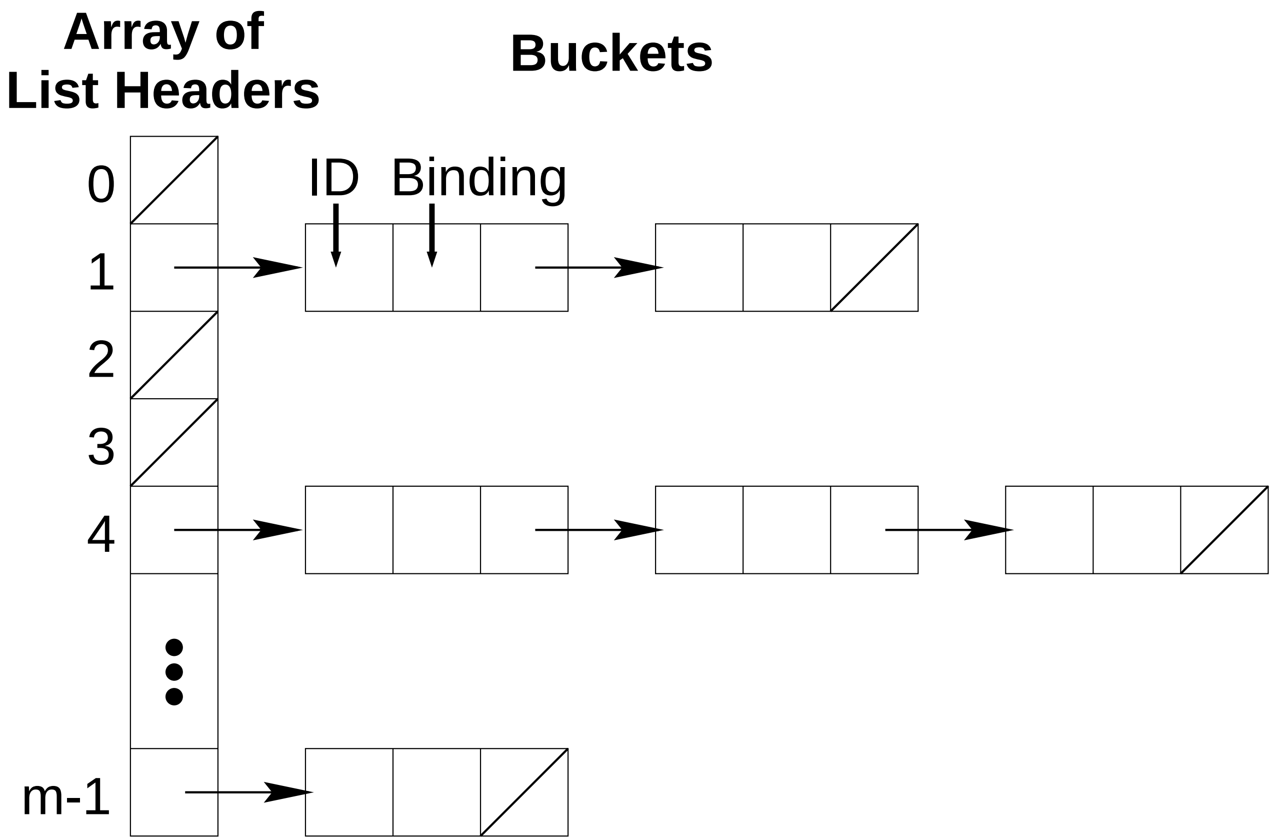
prefer most recent binding so as to implement scope rules of language

Imperative implementation style

- modify σ_1 to σ_2
- undo modification to get back to σ_1
- use *hash tables*
- $\sigma' = \sigma + \{a \to \tau\}$ Implement by inserting τ into hash table with key a

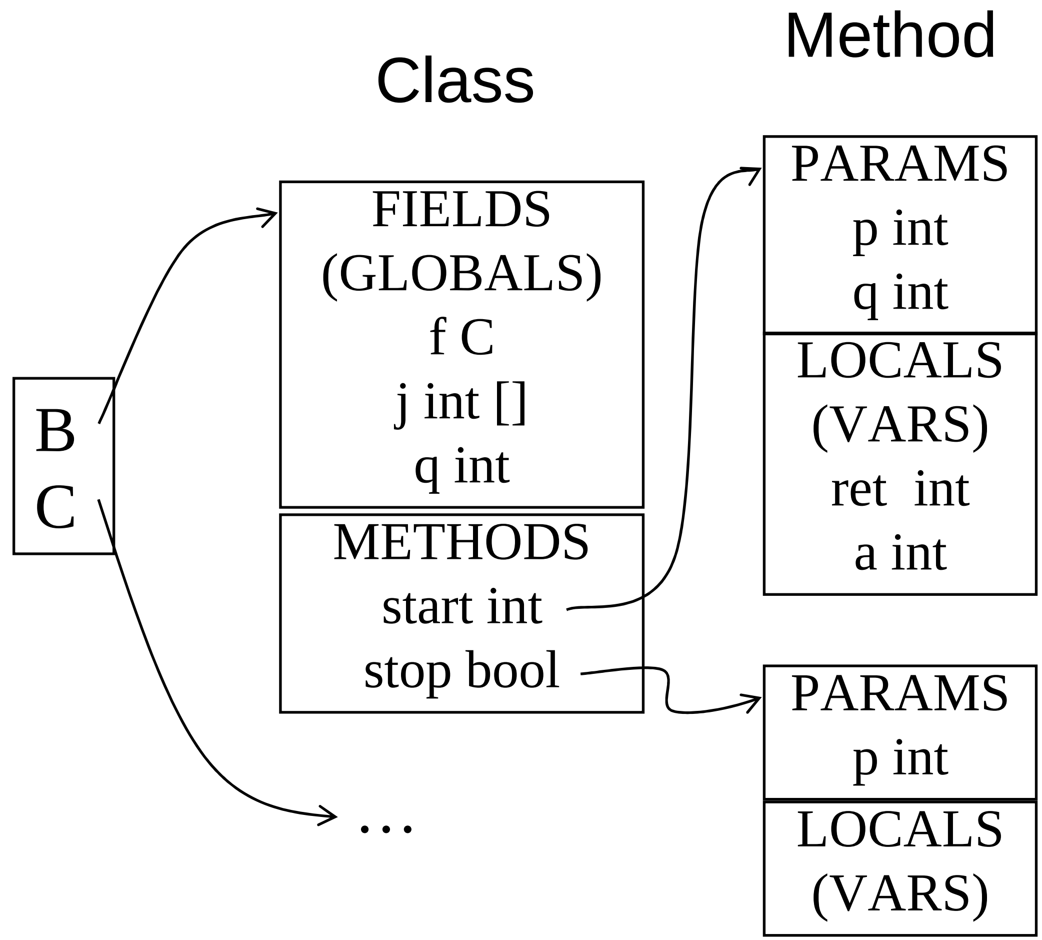
- simple hash table with *external chaining*: *i* th bucket = linked list of all elements whose keys hash to *i* mod `SIZE`
- `java.util.Hashtable` implements this for us, and a hashtable will be used for storing classes, and within them two hashtables will store the global variables (fields) and methods, and within methods a hashtable will store the local variables

Hashing with external chaining



Symbol table for MiniJava

```
class B {
  C f; int [] j; int q;
  public int start(int p, int q) {
    int ret; int a;
    ...
    return ret;
  }
  public boolean stop(int p) {
    …
    return false;
  }
}
class C { … }
```



MiniJava symbols and typechecking

- each variable name and formal parameter bound to its type
- each method name bound to its formal parameters, result type, and local variables
- each class name bound to its variable (field) and method declarations

- typechecking:
  - first build the symbol table
  - then typecheck the statements and expressions in the AST, consulting the symbol table for each identifier found

Symbol table implementation

```
class SymbolTable {
  public SymbolTable();
  public boolean addClass(String id, String parent);
  public Class getClass(String id);
  public boolean containsClass(String id);
  public Type getVarType(Method m, Class c, String id);
  public Method getMethod(String id, String classScope);
  public Type getMethodType(String id, String classScope);
  public boolean compareTypes(Type t1, Type t2);
}
```

- SymbolTable contains a hashtable of Class objects…

Symbol table implementation

- getVarType(Method m, Class c, String id)
  - in c.m, find variable id
  - may be…
    - local variable in method
    - parameter in formal parameters in method
    - variable in the class
    - variable in a parent class
- getMethod(), getMethodType()
  - may be defined in the parent classes
- compareTypes()
  - primitive types IntegerType, BooleanType, IntegerArrayType
  - IdentifierTypes (class types) stored as strings, returns true if identical or if equals a parent

Symbol table class Class

```
class Class {
  public Class(String id, String parent);
  public String getId();
  public Type type();
  public boolean addMethod(String id, Type type);
  public Method getMethod(String id);
  public boolean containsMethod(String id);
  public boolean addVar(String id, Type type);
  public Variable getVar(String id);
  public boolean containsVar(String id);
  public String parent();
}
```

- Class contains a hashtable of global variables (fields) and a hashtable of methods

Symbol table variable representation

```
class Variable {
  public Variable(String id, Type type);
  public String id();
  public Type type();
}
```

Symbol table method representation

```
class Method {
  public Method(String id, Type type);
  public String getId();
  public Type type();
  public boolean addParam(String id, Type type);
  public Variable getParamAt(int i);
  public boolean getParam(String id);
  public boolean containsParam(String id);
  public boolean addVar(String id, Type type);
  public Variable getVar(String id);
  public boolean containsVar(String id);
}
```

- Method contains a vector of parameters (formals) and a hashtable of (local) variables, and a return type

Implementation

- is by visitors, just like the pretty printer
  - interfaces Visitor is as before
  - a new visitor interface TypeVisitor is just the same but its methods return a Type
  - general classes DepthFirstVisitor and TypeDepthFirstVisitor implement these interfaces and traverse the AST depth-first visiting each node, but taking no action

- building the symbol table
  - BuildSymbolTableVisitor extends TypeDepthFirstVisitor, overriding methods so as to add classes, methods, vars, etc
- typechecking
  - TypeCheckVisitor extends DepthFirstVisitor overriding methods so as to check statements, TypeCheckExpVisitor extends TypeDepthFirstVisitor overriding methods so as to check expressions

BuildSymbolTableVisitor

Note that some checks are done as the table is built, eg for variable declarations (see below) a check is made on whether the variable is already declared

```
public class BuildSymbolTableVisitor extends TypeDepthFirstVisitor {
…
 private Class currClass;
 private Method currMethod;
…
 // Type t;
 // Identifier i;
  public Type visit(VarDecl n) {

     Type t =  n.t.accept(this);
     String id =  n.i.toString();
```

BuildSymbolTableVisitor continued

```
 if (currMethod == null){
        if (!currClass.addVar(id,t)){
            System.out.println(id + "is already defined in "
                                                + currClass.getId());
        }
 } else {
        if (!currMethod.addVar(id,t)){
            System.out.println(id + "is already defined in "
                                                + currClass.getId() + "." +
                                                currMethod.getId());
        }
  }
  return null;
}
```

TypeCheckVisitor

```
public class TypeCheckVisitor extends DepthFirstVisitor {

    static Class currClass;
    static Method currMethod;
    static SymbolTable symbolTable;

    public TypeCheckVisitor(SymbolTable s){
        symbolTable = s;
    }
...
```

TypeCheckVisitor continued

```
…
// Identifier i;
// Exp e;
public void visit(Assign n) {

  Type t1 = symbolTable.getVarType(currMethod,currClass,n.i.toString());
  Type t2 = n.e.accept(new TypeCheckExpVisitor() );

  if (symbolTable.compareTypes(t1,t2)==false){
        System.out.println("Type error in assignment to "+n.i.toString());
  }
}
…
```

TypeCheckVisitor continued

```
…
// Type t;
// Identifier i;
// FormalList fl;
// VarDeclList vl;
// StatementList sl;
// Exp e;
public void visit(MethodDecl n) {
  n.t.accept(this);
  String id = n.i.toString();
  currMethod = currClass.getMethod(id);
  Type retType = currMethod.type();
  for ( int i = 0; i < n.fl.size(); i++ ) { n.fl.elementAt(i).accept(this); }
  for ( int i = 0; i < n.vl.size(); i++ ) { n.vl.elementAt(i).accept(this); }
  for ( int i = 0; i < n.sl.size(); i++ ) { n.sl.elementAt(i).accept(this); }

  if (symbolTable.compareTypes(retType, n.e.accept(new TypeCheckExpVisitor()))==false) {
      System.out.println("Wrong return type for method "+ id);
  }
}
```

TypeCheckExpVisitor

```
public class TypeCheckExpVisitor extends TypeDepthFirstVisitor {
…

 // Exp e1,e2;
 public Type visit(Plus n) {
   if (! (n.e1.accept(this) instanceof IntegerType) ) {
       System.out.println("Left side of Plus must be of type integer");
   }

   if (! (n.e2.accept(this) instanceof IntegerType) ) {
       System.out.println("Right side of Plus must be of type integer");
   }

   return new IntegerType();
 }
…
```

Method Calls $e.i(el_1, el_2, \ldots)$

- Lookup method in the SymbolTable to get parameter list and result type
- Find i in class e
- The parameter types in the parameter list for the method must be matched against the actual arguments el_1, el_2, …
- Result type becomes the type of the method call as a whole

TypeCheckExpVisitor continued

```
…
// Exp e;
// Identifier i;
// ExpList el;
public Type visit(Call n) {
  if (! (n.e.accept(this) instanceof IdentifierType)){
      System.out.println("method "+ n.i.toString()+ "called  on something that is not a class or Object.");
  }

  String mname = n.i.toString();
  String cname = ((IdentifierType) n.e.accept(this)).s;
  Method calledMethod = TypeCheckVisitor.symbolTable.getMethod(mname,cname);

  for ( int i = 0; i < n.el.size(); i++ ) {
      Type t1 =null;
      Type t2 =null;

      if (calledMethod.getParamAt(i)!=null)
          t1 = calledMethod.getParamAt(i).type();
      t2 = n.el.elementAt(i).accept(this);
      if (!TypeCheckVisitor.symbolTable.compareTypes(t1,t2)){
          System.out.println("Type Error in arguments passed to " +cname+"." +mname);
      }
  }
  return TypeCheckVisitor.symbolTable.getMethodType(mname,cname);
}
```

What you should do now…

- read and digest chapter 5
  - you don’t need functional implementation styles or mutiple tables
- get ready further to develop the MiniJava typechecker