Semantic Analysis/Checking

Semantic analysis: the final part of *analysis* half of compilation

- afterwards comes *synthesis* half of compilation

Purposes:

- perform final checking of legality of input program, "missed" by lexical and syntactic checking
  - name resolution, type checking, `break` stmt in loop, ...
- "understand" program well enough to do synthesis
  - e.g. relate assignments to & references of particular variable

Symbol tables

Key data structure during semantic analysis, code generation

Stores info about names used in program

- a map (table) from names to info about them
  - each **symbol table entry** is a **binding**
- a declaration adds a binding to map
- a use of a name looks up binding in map
  - report a type error if none found

Example

```
class C {
   int x;
   boolean y;
   int f(C c) {
      int z;
      ...
      ... z ... c ... new C() ... x ... f(..) ...
   }
}
```


A bigger example

```
class C {
   int x;
   boolean y;
   int f(C c) {
      int z;
      ...
      {
         boolean x;
         C z;
         int f;
         .. z .. c .. new C() .. x .. f(..) ..
      }
      .. z .. c .. new C() .. x .. f(..) ..
   }
}
```

Nested scopes

Can have same name declared in different scopes

Want references to use closest textually-enclosing declaration
- static/lexical scoping, block structure
- closer declaration **shadows** declaration of enclosing scope

Simple solution:
- one symbol table per scope
- each scope's symbol table refers to its lexically enclosing scope's symbol table
  - root is the global scope's symbol table
- look up declaration of name starting with nearest symbol table, proceed to enclosing symbol tables if not found locally

All scopes in program form a tree

Name spaces

Sometimes can have same name refer to different things, but still unambiguously

Example:

```
class F {
  int F(F F) {
    // 3 different F's are available here!
    ... new F() ...
    ... F = ...
    ... this.F(...) ...
  }
}
```

In MiniJava: three **name spaces**
- classes, methods, and variables

We always know which we mean for each name reference, based on its syntactic position

Simple solution:
symbol table stores a separate map for each name space

Information about names

Different kinds of declarations store different information about their names
- must store enough information to be able to check later references to the name

A variable declaration:
- its type
- whether it's `final`, etc.
- whether it's `public`, etc.
- (maybe) whether it's a local variable, an instance variable, a global variable, or ...

A method declaration:
- its argument and result types
- whether it's `static`, etc.
- whether it's `public`, etc.

A class declaration:
- its class variable declarations
- its method and constructor declarations
- its superclass

Generic typechecking algorithm

To do semantic analysis & checking on a program, recursively typecheck each of the nodes in the program's AST, each in the context of the symbol table for its enclosing scope
- on the way down, create any nested symbol tables & context needed
- recursively typecheck child subtrees
- on the way back up, check that the children are legal in the context of their parents

Each AST node class defines its own `typecheck` method, which fills in the specifics of this recursive algorithm

Generally:
- declaration AST nodes add bindings to the current symbol table
- statement AST nodes check their subtrees
- expression AST nodes check their subtrees and return a result type

MiniJava typechecker implementation

In `SymbolTable` subdirectory:

Various `SymbolTable` classes, organized into a hierarchy:

```
SymbolTable
   GlobalSymbolTable
   NestedSymbolTable
      ClassSymbolTable
      CodeSymbolTable
```

Support the following operations (and more):

- `declareClass`, `lookupClass`
- `declareInstanceVariable`, `declareLocalVariable`, `lookupVariable`
- `declareMethod`, `lookupMethod`

Class, variable, and method information

`lookupClass` returns a `ClassSymbolTable`

- includes all the information about the class's interface

`lookupVariable` returns a `VarInterface`

- stores the variable's type

A hierarchy of implementations:

```
VarInterface
   LocalVarInterface
   InstanceVarInterface
```

`lookupMethod` returns a `MethodInterface`

- stores the method's argument and result types

Some key AST typecheck operations

```
void Program.typecheck()
   throws TypecheckCompilerExn;
```

- typecheck the whole program

```
void Stmt.typecheck(CodeSymbolTable)
   throws TypecheckCompilerExn;
```

- typecheck a statement in the context of the given symbol table

```
ResolvedType Expr.typecheck(CodeSymbolTable)
   throws TypecheckCompilerExn;
```

- typecheck an expression in the context of the given symbol table, returning the type of the result

Forward references

Typechecking class declarations is tricky: need to allow for **forward references** from the bodies of earlier classes to the declarations of later classes

```
class First {
   Second next;  // have to allow this forward reference
   int f() {
      ... next.g() ...  // and this forward reference
   }
}
class Second {
   First prev;
   int g() {
      ... prev.f() ...
   }
}
```

Supporting forward references

Simple solution:
typecheck a program's class declarations in multiple passes

- first pass: remember all class declarations
  {`First` → class{?}, `Second` → class{?}}
- second pass: compute interface to each class, checking class types in headers
  {`First` → class{`next`:`Second`},
  `Second` → class{`prev`:`First`}}
- third pass: check method bodies, given interfaces

```
void ClassDecl.declareClass(GlobalSymbolTable)
   throws TypecheckCompilerExn;
```

- declare the class in the global symbol table

```
void ClassDecl.computeClassInterface()
   throws TypecheckCompilerExn;
```

- fill out the class's interface, given the declared classes

```
void ClassDecl.typecheckClass()
   throws TypecheckCompilerExn;
```

- typecheck the body of the class, given all classes' interfaces

An example typechecking operation

```
class VarDeclStmt {
   String name;
   Type type;

   void typecheck(CodeSymbolTable st)
         throws TypecheckCompilerException {
      st.declareLocalVar(type.resolve(st), name);
   }
}
```

`resolve` checks that a syntactic type expression is a legal type, and returns the corresponding resolved type

`declareLocalVar` checks for duplicate variable declaration in this scope

An example typechecking operation

```
class AssignStmt {
   String lhs;
   Expr rhs;

   void typecheck(CodeSymbolTable st)
         throws TypecheckCompilerException {
      VarInterface lhs_iface = st.lookupVar(lhs);
      ResolvedType lhs_type = lhs_iface.getType();
      ResolvedType rhs_type = rhs.typecheck(st);
      rhs_type.checkIsAssignableTo(lhs_type);
   }
}
```

`lookupVar` checks that the name is declared as a var

`checkIsAssignableTo` verifies that an expression yielding the rhs type can be assigned to a variable declared to be of the lhs type

- initially, rhs type is equal to or a subclass of lhs type

An example typechecking operation

```
class IfStmt {
   Expr test;
   Stmt then_stmt;
   Stmt else_stmt;

   void typecheck(CodeSymbolTable st)
         throws TypecheckCompilerException {
      ResolvedType test_type = test.typecheck(st);
      test_type.checkIsBoolean();
      then_stmt.typecheck(st);
      else_stmt.typecheck(st);
   }
}
```

`checkIsBoolean` checks that the type is a boolean

An example typechecking operation

```
class BlockStmt {
   List<Stmt> stmts;

   void typecheck(CodeSymbolTable st)
         throws TypecheckCompilerException {
      CodeSymbolTable nested_st =
         new CodeSymbolTable(st);
      foreach Stmt stmt in stmts {
         stmt.typecheck(nested_st);
      }
   }
}
```

(Garbage collection will reclaim `nested_st` when done)