Compilation: PA3
Fall 2014

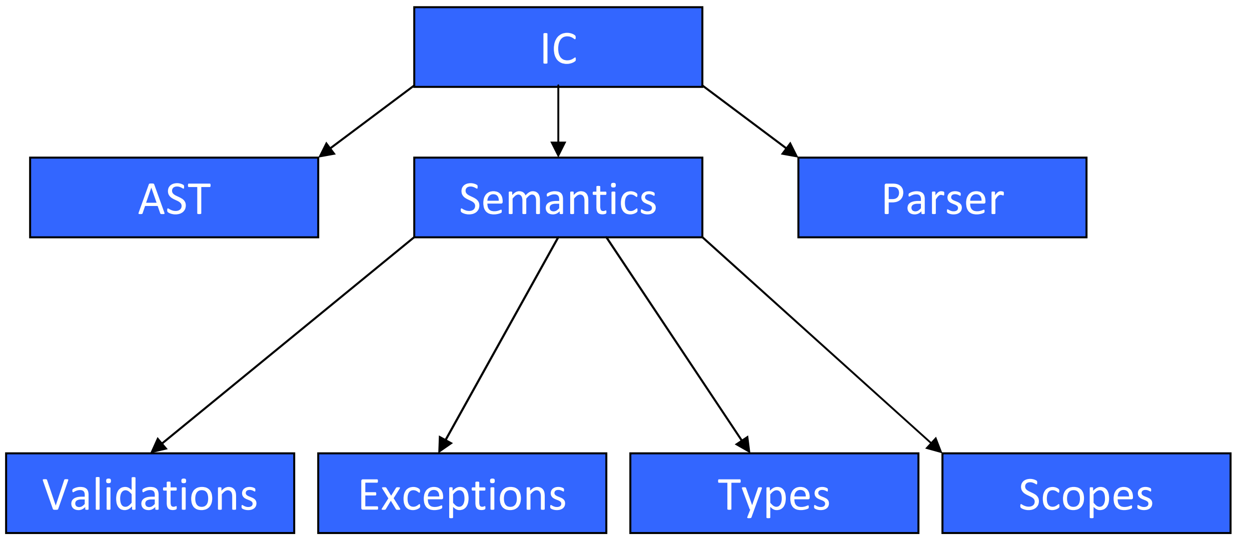
Ice Coffee Semantics: Documentation

Ori Lentzitzky	200446094
Idan Shaby	200789691
Guy Engel	300455672

Package Structure

In PA3, we've built upon the previous assignment (PA2), and placed our new code with in the *Semantics* sub-package.

The following diagram describes the hierarchy of the project:



IC.Semantics is the main package of the semantic checks. *SemanticCheck*s class within the package has a method run(), which runs all semantic checks (validations) in sequence. It stops once an error has been found and prints the errror to the screen.

IC.Semantics.Scopes package holds the Symbol Table ADT structure.

IC.Semantics.Types package holds the Type Tabel ADT structure.

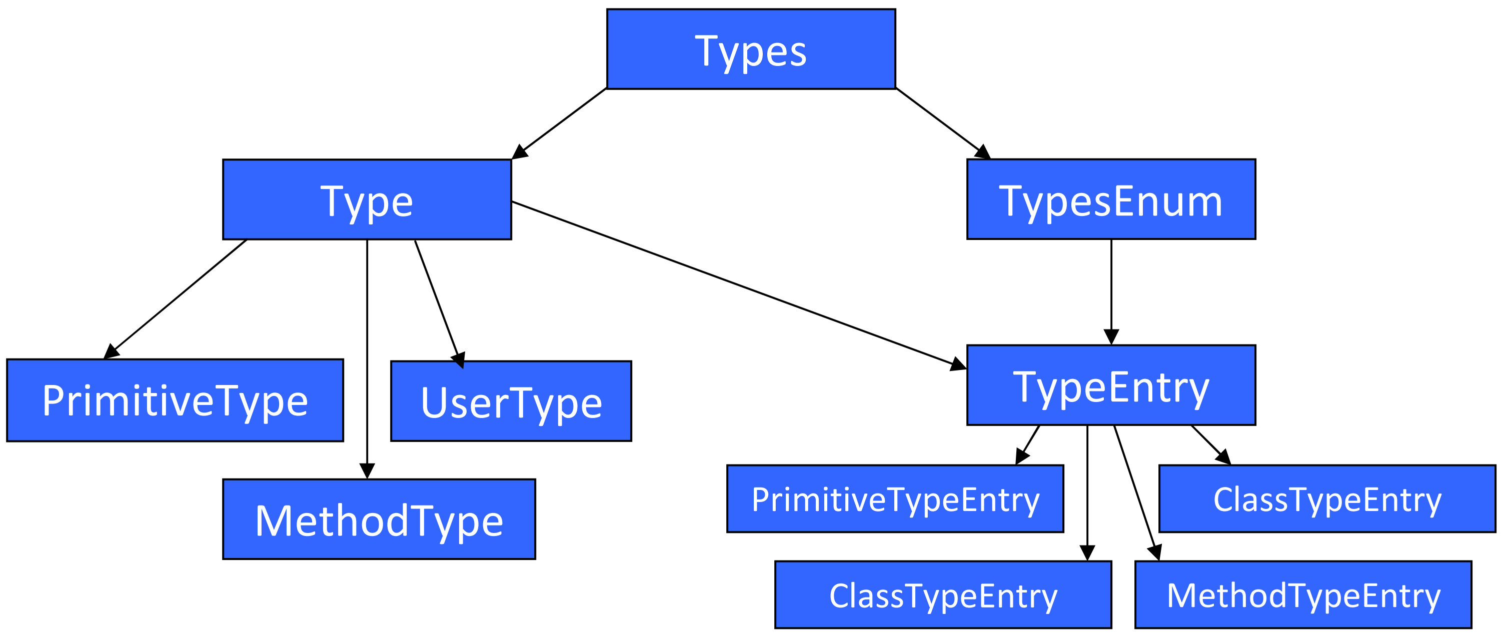
IC.Semantics.Validations package consists of the various validations / semantic checks performed on the AST, the Symbol Table and the Type Table.

Symbol (Scope) And Type Tables

Type Tables

As a whole, the type table implementation is found with in the **IC.Semantics.Types** package.

Structure



TypeEntry

TypeEntry objects represent a single entry in the type table. They consist of a TypeEnum (Primitive, Class, Array, Method; each one given a precedence to facilitate with printing), the string representation and a serial integer. Each entry in the table receives a different id, in ascending order of their insertion into the table (the order in which we encounter them during construction).

When printing the type table, the entries are sorted first by precedence and when precedences match, the secondary sort is performed by the entries' ids.

TypeEntry is abstract and each type of entry has a corresponding derived class with its' added attributes as needed (for example, *ClassTypeEntry* keeps a reference to the *ICClass* node of the AST parse tree linked to that class).

Type Table

The *type table* is basically a map object between types' string representation to their corresponding TypeEntry object.

Adding type to the type table

Adding a new type to the type table is only possible if the type doesn't already exist within the table (no entry with the same string represenation of the type).

When adding an array to the type table, the type table first checks if the standard type is defined (meaning, for A[][][], if there exists a type A in the table). If not, it adds it (assuming that A is a user-type; this is a valid assumption since construction makes it mandatory for primitives to be added first thing, and methods can be "array-ed"). Then, all smaller dimensions (i.e. A[] and A[][] for the aboved example) are checked to see if they exist within the table, and if not added. Only once all of these checks and additions were made, the type is added to the table.

Scopes Tables (Symbol Tables)

We've dubbed the symbol table "Scopes Table" to make the code a bit more readable (making clear that each scope has its' own symbol table).

The scopes implementation is found within **IC.Semantics.Scopes** package.

Symbol

Symbol class represents an entry of the scopes / symbol table.

ID	Type	Kind	Node

- ID – a string representation of the symbol.
- Type – PrimitiveType or UserType from the **IC.Semantics.Types** package.
    - Arrays are represented like in the AST with a field *dimension*.
    - MethodType is not used within the symbol / scopes table. For a method entry, the type of the method is its' return type.
- Kind – an enum detailing one of the possible kinds of entries:
    - Field
    - Formal
    - Class
    - Local Variable
    - Static Method
    - Virtual Method
- Node – a reference to the ASTNode where the corresponding entry was created (for methods, it will the Method node; for Class, it will be the ICClass node, and so on).

Scope

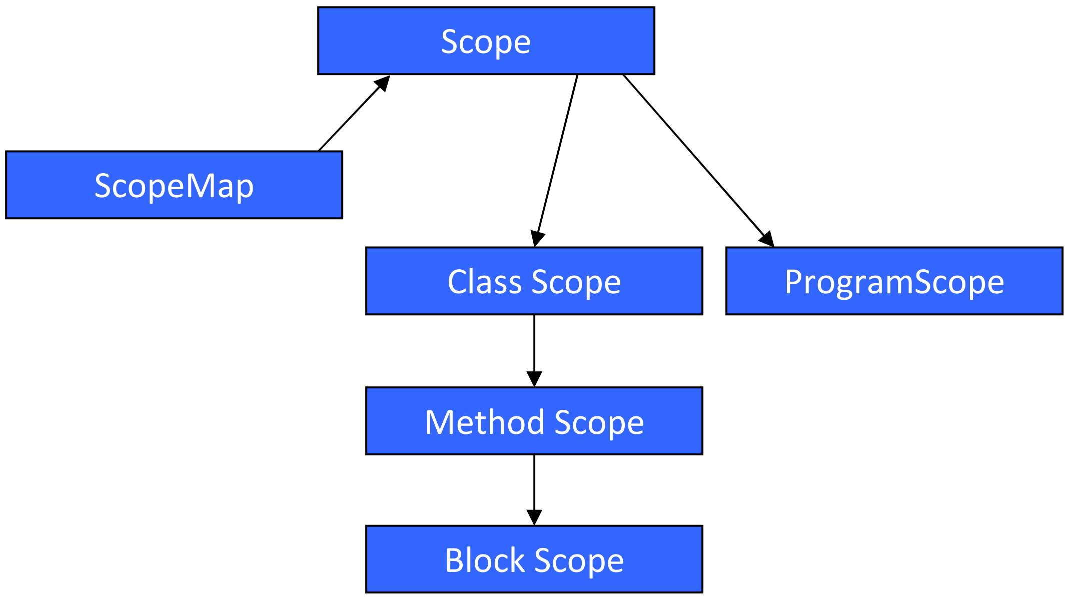
Scope class represents the scopes table. It is an abstract class maintaing a mapping between string representation of symbols to their symbol values.

Scope keeps a reference to its' parent scope in the scope hierarchy, as well as a reference to each one of its' children.

Scope Hierarchy

Unlike the *Type Table*, which is one and only, there are many scopes tables – one for each scope, as mentioned above. The scopes are organized into a graph structure (for a valid program, the graph would be a connected tree where G=(V,E) holds |E|=|V|-1).

The scope objects themselves derive from each other (as the following diagram shows).



ProgramScope is the ***global symbol table***. Its' entries are the program classes, its' children are the classes which don't extend any other classes.

ClassScope of a class which inhierts from another class, is placed directly under that class (as its' child, not its' brother). For example, if class B extends class A and class A does not extend any [specific] class, then class A's scope will be placed under the global scope and class B's scope will be placed under class A's scope.

All methods of a class have their own *MethodScope* and are placed under the appropriate *ClassScope* in which they are defined.

All blocks ({ }) within a method have their own *BlockScope*, which is placed under the method. Any scopes defined within another scope, are placed under that scope and so on.

Adding symbol to the scopes table

When adding a symbol to the scopes table, an exception is thrown if a symbol by that name is already defined in that scope (only one scope with given name is allowed per scope). The exception to that is when defining both a virtual and a static method of the same name in the same *ClassScope* – both can be defined without a problem.

If a field is defined in a superclass, and a field with a similar name is defined in a derived class, both are added to the symbol table without throwing an exception. However, the IC Specification clearly states this is illegal. Once construction has finished, it scans over the scopes of dervied classes. If a field is found to override an existing field in a superclass, now an exception is thrown.

Modifications to the AST

In order to support easy access between the AST nodes and the type and scopes table, we added two new fields to the *ASTNode* class:

```java
private Scope enclosingScope;
private Type nodeType;
```

Each *ASTNode* is now linked directly to its' corresponding scope. That is, every *ICClass* is linked to the *ClassScope*, every method to the *MethodScope*, and every *StatementsBlock* to the *BlockScope*. Except for methods (which have their own scope, but are entries in the *ClassScope* they are defined in), each node is linked directly to the scopes table it is defined in (fields in *ClassScope*, formals in *MethodScope*, and so on).

The *nodeType* field links every node to its' corresponding semantic type.

Remember!: Each semantic type is defined only once in the types table (has only one corresponding entry).

Most of the higher *ASTNode*s do not have a type (nodeType == null). These are mostly statements who have no real type. However, fields, formals, expressions, methods and so on have a type set for them.

→ Every node is linked to a symbol table. Most nodes are linked to a semantic type.

Construction

IC.Semantics.ScopesTypesBuilder is the class responsible for building both the scopes and type tables. They are built hand-in-hand.

Scopes Table Construction

Built using the AST parse tree *Visitor* interface.

- Start with Program node. Create a *ProgramScope* and link it to the node.
- Visit the nodes down the tree.
- For each visited node:
  - If Class, Method or StatementsBlock, create a new scope object of the corresponding type and link it to node.
  - For other nodes, create an entry for them in the scope and link them to the table (i.e., for field add it to the *ClassScope* and link node to scope, etc).

Types Table Construction

- Start with the Program node.
- Add all primitives to the type table.
- Add main method signature to the type table (to match required printing of type table).
- Visit the nodes down the tree.
  - Whenever you encounter a type, add it to the type table (if already exists, type table will ignore it).
  - For Method nodes, add MethodType to the type table.

Semantic Checks

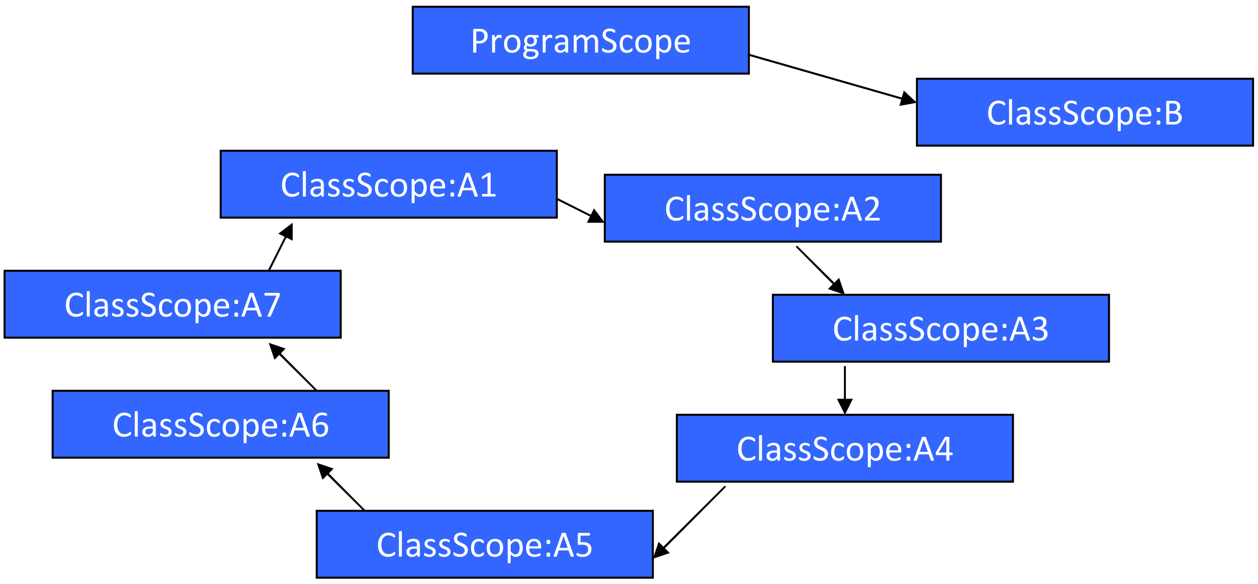
All semantic checks are performed traversing the AST parse tree (excpet for the first check – *NonCircularScopesValidation*). All use the *Visitor* design pattern.

Required semantic validations

NonCircularScopesValidation

Ensures that the class hierarchy is in fact a tree. The could be performed on the type table, however we found it much easier to simply check the scopes hierarchy. The check is made using the *Visitor* design pattern, traversing over the scopes objects.

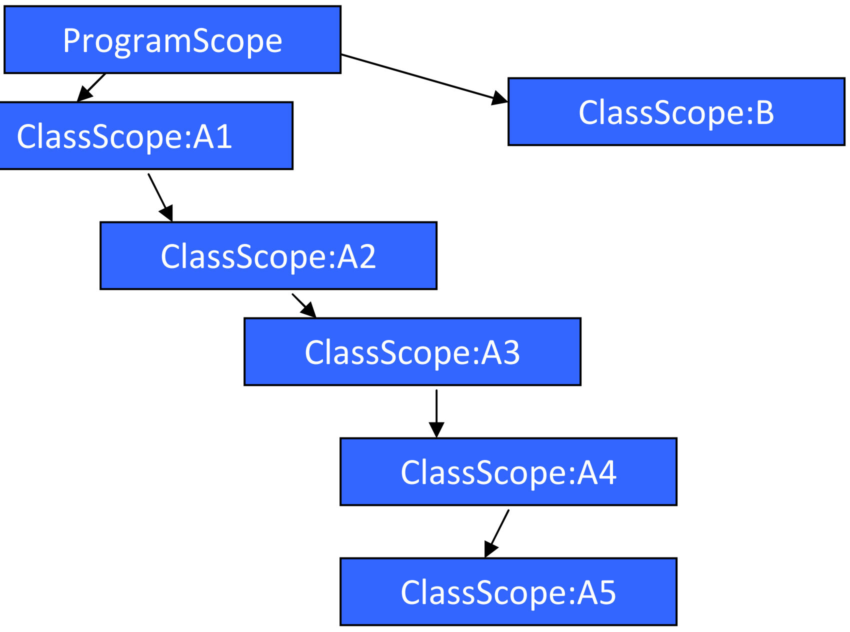
Whenever a class extends another class, it is placed directly under it in the scopes table. Suppose we have a circular definition of classes: A1…An. Assume without loss of generality, A1 extends A2, A2 extends A3 and so on until An extends A1 (thus completing a circle). In reality, none of these classes is a direct child of the *ProgramScope*, even though all of them are entries in the *ProgramScope*.



Algorithm for validation works as follows:

- Let A be the set of all classes defined within the *ProgramScope* table.
- Let B be the set of direct descendants of the *ProgramScope* table.
- Let C = A – B, be the set of all classes defined within the *ProgramScope* table that are not direct descendants of the table.
- Let maxCount = |C|.
- If maxCount = 0, no circular dependencies.
- Otherwise, visit each element of C and count how many times you've visited it. If an element has been visited (maxCount + 1) times, throw exception (there's a circle in the hierarchy).

Why (maxCount+1) times? Because worst case would be a total of maxCount classes deriving from each other not in a circle. So the lowest class will be visited maxCount times (in the following diagram, A5 will be visited once from a visit of A1 to its' descendants, once from a visit of A2 to its' descends and so on, a total of 5 times. So if we've visited it 6 times, it must mean there's a circle in the hierarchy).



DeclarationValidation

Validates that each field / formal / variable / method / class have been declared when used. For instance: for

```
a = foo(x, y, z, bar(w));
```

checks that variables a, x, y, z, and w were previously declared in the current scope (or one of its' parents) and that a method names foo and a method named bar exists as well. Moreove, it checks that foo expects 4 arguments and bar expects 1 argument. **Types are not checked yet!**

Since in the course forum, the TA has instructed us to support calls to static methods with the class identifier (as long as they exist within the same scope, i.e. the same class as the call), this requires a bit of a twicking.

The CFG of Ice-Coffee and the parse tree of PA2 cannot support this, since it is impossible to know whether or not this is a static call or a virtual call. Therefore, we consider every virtual call that *is **not** external* to possibly be a static call. Meaning the call `foo(x,y)` can be either virtual or static, but the call `A.foo(x,y)` is definitely static and the call `a.foo(x,y)` is definitely virtual.

We implement this by searching for both a static and a virtual method with the given name. If we only found one, then we choose the one we found to be the one the programmer must have intended. If we found none, well, we notify the programmer of his mistake. However, is we found both then we look at the number of formals they expect. If both expect a different number of formals and one of those numbers matches the number of arguments in the call, we choose the one that matches. Otherwise, we notify the programmer we cannot understand which one he wanted to call and he should clarify that (for virtual, do `this.foo(x,y)` and for static do `A.foo(x, y)` even if he's writing in the scope of the class A).

Examples:

```
class A {
      static void foo(int x, int y);
      void foo(int x, int y, int z);
      int doSomething() {
            foo(1, 2);
      }
}
```

Here, the static foo will be called.

```
class A {
      static void foo(int x, int y);
      void foo(int x, int y, int z);
      int doSomething() {
            foo(1, 2, 3);
      }
}
```

Here, the virtual foo will be called.

```
class A {
      static void foo(int x, int y);
      void foo(int x, int y);
      int doSomething() {
            foo(1, 2);
      }
}
```

Here, an exception will be thrown.

```
class A {
      static void foo(int x, int y);
      void foo(int x, int y, int z);
      int doSomething() {
            foo(1);
      }
}
```

Here, an exception will be thrown (no method foo that expects one argument was declared).

Whenever inside a static method implementation, *DeclarationValidation* only searches for local variables and formals. Static method cannot access the instance scope and fields are specific for each instantiation of the object.

TypesValidation

Checks legality and validity of types according to IC typing rules (Section 15).

- Variables are assigned expression evaluating to the right type.
- If types don't correspond, an inheritance relation exists between assigned value to assigned variable.
- Methods' return statements are of the correct return type the method declared of in its' signature.
- Method call arguments match to the method signature.
- Override method maintain types of overriden method (may only replace types by their dervied types).
- Condition expressions are evaluated to boolean types.
- Integer literal is a 32-bit signed integer (ranges from -2^{31} to $2^{31}-1$).
- And so on.

SingleMainValidation

Validates that in the entire program, including the library class (if supplied), only one method exists by the name *main*. Moreover, validates that said method has the correct signature. If no method was found to match, or more than one was found, an exception is thrown.

ControlStatementsValidations

Validates that *break* and *continue* statements appear only within loop (while loop is the only kind of loop that exists in the IC language).
Also, validates that *this* keyword is not used within static method scope.

Library Class

The required check that supplied library class actually is called Library (i.e. is defined as "class Library { ") was already done in PA2.

Bonus Checks

LocalVariableInitializedValidation

Validates that every local variable used was initialized with a value previously.

Examples:

```
x = x + 1;
```

We've previously checked x was declared, but now we check that it also has been initalized. So for:

```
int x;
x = x + 1;
```

We'll throw an exception, but for:

```
int x = 0;
x = x + 1;
```

we won't.

Moreover, for a complex statement such as:

```
x = a.foo(i+j, x-3, y);
```

We check that a, i, j and x have been initiazlied. We do **not** check that y has been initiazlied (since it is only passed but not used in this context).

For arrays, we only check that they have been initiazlied with the new keyword, not that their contents are initialized (according to the spec, arr[i] is checked not to be null during run-time, not compile time). We also do not check for index out of bounds for array.

Note that for:

```
int x;
if (checkSomething())
      x = 3;
x = x + 1;
```

would throw an exception because not all code paths initialize x. The same is true is we replace if with a while statement. However:

```
int x;
if (checkSomething())
      x = 3;
else
      x = 2;
x = x + 1;
```

would be okay and not throw an exception.

```
int x;
int y;
if (checkSomething())
      x = 3;
else
      y = 1;
x = x + 1;
```

This would still throw an exception because x hasn't been initiazlied by all paths.

NonVoidAlwaysReturnsValidation

Validates that every non-void return method returns a value on every one of its' control paths.

We do so by looking for a return statement in the out most scope of the method (including within block statements that are not children to if statements or while statements). If we found a return statement in the method, then it will always return a value. If not, then only if the method has a branching statement and contains both an if and an else part, and both parts return a value, we can safely say it returns on every control paths.

Returns inside while loops are ignored since we may not have branched into the while loop and therefore not all code paths return a value.

Miscellaneous Checks

To help us and to faciliate, we also provide the following checks, though not completely (not all cases may have been caught):

Dead code

For one of two instances:

When code continues after return statements:

```
return;
//some code
```

Everything after return would be unreachable (dead code).

```
if (checkSomething()) {
      ...
      return 1;
} else {
      ...
      return 0;
}
//some code
return 1;
```

Everything after the if/else part is dead code and is unreachable since method will encounter a return statement before that.

The other instance is where we can (and we don't always can) predict the outcome of a branch (usually before if/else/while statements and if all local variables were initialzied with literals) :

```
int x = 2;
if (x > 2)
      return;
//some code
```

Everything after the if is dead code in this instance.

More of the same

To the same extent, whenever we could (usually at the beginning of methods and scopes, before if/else/while statements have been seen and when local variables have been initialized with literals) we may search for:

```
int x = 1;
int y = 0;
int z = x / y;
```

throws an error because of division by zero.

```
int x = -2;
int[] y = new int[x];
```

throw an error because array is initialized with negative size.

Testing

We've built a few test programs to test out various scenarios for each one of the validations. Some code excerpts were given before.

Moreover, we ran all the examples and the Quicksort from the previous assignment to make sure we pass all semantic checks.

For each test that ran okay, we changed a few things to cause semantic errors and made sure we got the expected exception / error.

For instance, for:

```
class A { }
class B extends A { }
class C {
      A foo(int x);
}
class D {
      B foo(int x);
}
```

We didn't see any error, but for:

```
class A { }
class B extends A { }
class C {
      A foo(int x);
}
class D {
      B foo(A x);
}
```

We got a semantic error for mismatch type between int and A in formal of overriden method.

And so on….