06-0: Syntax Errors/Semantic Errors

- A program has syntax errors if it cannot be generated from the Context Free Grammar which describes the language
- The following code has no syntax errors, though it has plenty of semantic errors:

```java
void main() {
    if (3 + x - true)
        x.y.z[3] = foo(z);
}
```

- Why don’t we write a CFG for the language, so that all syntactically correct programs also contain no semantic errors?

06-1: Syntax Errors/Semantic Errors

- Why don’t we write a CFG for the language, so that all syntactically correct programs also contain no semantic errors?
- In general, we can’t!
  - In simpleJava, variables need to be declared before they are used
  - The following CFG:
    - \( L = \{ww|w \in \{a, b\}\} \)
    - is not Context-Free – if we can’t generate this string from a CFG, we certainly can’t generate a simpleJava program where all variables are declared before they are used.

06-2: JavaCC & CFGs

- JavaCC allows actions – arbitrary Java code – in rules
- We could use JavaCC rules to do type checking
- Why don’t we?

06-3: JavaCC & CFGs

- JavaCC allows actions – arbitrary Java code – in rules
- We could use JavaCC rules to do type checking
- Why don’t we?
  - JavaCC files become very long, hard to follow, hard to debug
  - Not good software engineering – trying to do too many things at once

06-4: Semantic Errors/Syntax Errors

- Thus, we only build the Abstract Syntax Tree in JavaCC (not worrying about ensuring that variables are declared before they are used, or that types match, and so on)
- The next phase of compilation – Semantic Analysis – will traverse the Abstract Syntax Tree, and find any semantic errors – errors in the meaning (semantics) of the program
• Semantic errors are all compile-time errors other than syntax errors.

06-5: **Semantic Errors**

• Semantic Errors can be classified into the following broad categories:

**Definition Errors**

• Most strongly typed languages require variables, functions, and types to be defined before they are used with some exceptions –
  • Implicit variable declarations in Fortran
  • Implicit function definitions in C

06-6: **Semantic Errors**

• Semantic Errors can be classified into the following broad categories:

**Structured Variable Errors**

• \(x.y = A[3]\)
  • \(x\) needs to be a class variable, which has an instance variable \(y\)
  • \(A\) needs to be an array variable
• \(x.y[z].w = 4\)
  • \(x\) needs to be a class variable, which has an instance variable \(y\), which is an array of class variables that have an instance variable \(w\)

06-7: **Semantic Errors**

• Semantic Errors can be classified into the following broad categories:

**Function and Method Errors**

• \(\text{foo}(3, \text{true}, 8)\)
  • \(\text{foo}\) must be a function which takes 3 parameters:
    • integer
    • boolean
    • integer

06-8: **Semantic Errors**

• Semantic Errors can be classified into the following broad categories:

**Type Errors**

• Build-in functions – \(/, *, ||, &&\), etc. – need to be called with the correct types
  • In simpleJava, \(+, -, *, /\) all take integers
  • In simpleJava, \(\|\) 
  • Standard Java has polymorphic functions & type coercion

06-9: **Semantic Errors**

• Semantic Errors can be classified into the following broad categories:
• Type Errors
  • Assignment statements must have compatible types
  • When are types compatible?

06-10: Semantic Errors

• Semantic Errors can be classified into the following broad categories:
  • Type Errors
    • Assignment statements must have compatible types
      • In Pascal, only Identical types are compatible

06-11: Semantic Errors

• Semantic Errors can be classified into the following broad categories:
  • Type Errors
    • Assignment statements must have compatible types
      • In C, types must have the same structure
      • Coerceable types also apply

```
struct {
  int x;
  char y;
} var1;

struct {
  int z;
  char x;
} var2;
```

06-12: Semantic Errors

• Semantic Errors can be classified into the following broad categories:
  • Type Errors
    • Assignment statements must have compatible types
      • In Object oriented languages, can assign subclass value to a superclass variable

06-13: Semantic Errors

• Semantic Errors can be classified into the following broad categories:
  • Access Violation Errors
    • Accessing private / protected methods / variables
    • Accessing local functions in block structured languages
    • Separate files (C)

06-14: Environment

• Much of the work in semantic analysis is managing environments
  • Environments store current definitions:
    • Names (and structures) of types
    • Names (and types) of variables
• Names (and return types, and number and types of parameters) of functions

• As variables (functions, types, etc) are declared, they are added to the environment. When a variable (function, type, etc) is accessed, its definition in the environment is checked.

06-15: **Environments & Name Spaces**

• Types and variables have different name spaces in simpleJava, C, and standard Java:

  simpleJava:

  ```java
  class foo {
    int foo;
  }
  void main() {
    foo foo;
    foo = new foo();
    foo.foo = 4;
    print(foo.foo);
  }
  ```

06-16: **Environments & Name Spaces**

• Types and variables have different name spaces in simpleJava, C, and standard Java:

  C:

  ```c
  #include <stdio.h>
  typedef int foo;
  int main() {
    foo foo;
    foo = 4;
    printf("%d", foo);
    return 0;
  }
  ```

06-17: **Environments & Name Spaces**

• Types and variables have different name spaces in simpleJava, C, and standard Java:

  Java:

  ```java
  class EnviornTest {
    static void main(String args[]) {
      Integer Integer = new Integer(4);
      System.out.print(Integer);
    }
  }
  ```

06-18: **Environments & Name Spaces**
• Variables and functions in C share the same name space, so the following C code is **not** legal:

```c
int foo(int x) {
    return 2 * x;
}

int main() {
    int foo;
    printf("%d\n", foo(3));
    return 0;
}
```

• The variable definition `int foo;` masks the function definition for `foo`.

**06-19: Environments & Name Spaces**

• Both standard Java and simpleJava use different name spaces for functions and variables.

• Defining a function and variable with the same name will not confuse Java or simpleJava in the same way it will confuse C.

  • *Programmer* might still get confused ...

**06-20: simpleJava Environments**

• We will break simpleJava environment into 3 parts:

  • **type environment** Class definitions, and built-in types int, boolean, and void.

  • **function environment** Function definitions – number and types of input parameters and the return type.

  • **variable environment** Definitions of local variables, including the type for each variable.

**06-21: Changing Environments**

```c
int foo(int x) {
    boolean y;

    x = 2;
    y = false;
    /* Position A */
    { int y;
        boolean z;

        y = 3;
        z = true;
        /* Position B */
    }
    /* Position C */
}
```

**06-22: Implementing Environments**

• Environments are implemented with Symbol Tables.
Symbol Table ADT:

- Begin a new scope.
- Add a key / value pair to the symbol table
- Look up a value given a key. If there are two elements in the table with the same key, return the most recently entered value.
- End the current scope. Remove all key / value pairs added since the last begin scope command

06-23: Implementing Symbol Tables

- Implement a Symbol Table as a list
  - Insert key/value pairs at the front of the list
  - Search for key/value pairs from the front of the list
  - Insert a special value for “begin new scope”
  - For “end scope”, remove values from the front of the list, until a “begin scope” value is reached

06-24: Implementing Symbol Tables

![Diagram of Symbol Tables]

06-25: Implementing Symbol Tables

- Implement a Symbol Table as an open hash table
  - Maintain an array of lists, instead of just one
  - Store (key/value) pair in the front of list[hash(key)], where hash is a function that converts a key into an index
  - If:
    - The hash function distributes the keys evenly throughout the range of indices for the list
    - # number of lists = $\Theta$ (# of key/value pairs)
    - Then inserting and finding take time $\Theta(1)$

06-26: Hash Functions
long hash(char *key, int tableSize) {
    long h = 0;
    long g;
    for (;*key;key++) {
        h = (h << 4) + *key;
        g = h & OxF0000000;
        if (g) h ^= g >> 24
        h &= g
    }
    return h % tableSize;
}

06-27: Implementing Symbol Tables

- What about beginScope and endScope?
- The key/value pairs are distributed across several lists – how do we know which key/value pairs to remove on an endScope?

06-28: Implementing Symbol Tables

- What about beginScope and endScope?
- The key/value pairs are distributed across several lists – how do we know which key/value pairs to remove on an endScope?
  - If we knew exactly which variables were inserted since the last beginScope command, we could delete them from the hash table
  - If we always enter and remove key/value pairs from the beginning of the appropriate list, we will remove the correct items from the environment when duplicate keys occur.
  - How can we keep track of which keys have been added since the last beginScope?

06-29: Implementing Symbol Tables

- How can we keep track of which keys have been added since the last beginScope?
- Maintain an auxiliary stack
  - When a key/value pair is added to the hash table, push the key on the top of the stack.
  - When a “Begin Scope” command is issued, push a special begin scope symbol on the stack.
  - When an “End scope” command is issued, pop keys off the stack, removing them from the hash table, until the begin scope symbol is popped

06-30: Type Checking

- Built-in types ints, floats, booleans, doubles, etc. simpleJava only has the built-in types int and boolean
- Structured types Collections of other types – arrays, records, classes, structs, etc. simpleJava has arrays and classes
- Pointer types int *, char *, etc. Neither Java nor simpleJava have explicit pointers – no pointer type. (Classes are represented internally as pointers, no explicit representation)
- Subranges & Enumerated Types C and Pascal have enumerated types (enum), Pascal has subrange types. Java has neither (at least currently – enumerated types may be added in the future)
06-31: **Built-In Types**

- No auxiliary information required for built-in types int and boolean (an int is and int is an int)
- All types will be represented by pointers to type objects
- We will only allocate *one* block of memory for *all* integer types, and *one* block of memory for *all* boolean types

06-32: **Built-In Types**

```c
void main() {
    int x;
    int y;
    boolean a;
    boolean b;

    x = y;  // Type Error
    x = a;  /* Type Error */
}
```

06-33: **Built-In Types**

![Type Environment Diagram](image)

06-34: **Class Types**

- For built-in types, we did not need to store any extra information.
- For Class types, what extra information do we need to store?

06-35: **Class Types**

- For built-in types, we did not need to store any extra information.
- For Class types, what extra information do we need to store?
  - The name and type of each instance variable
  - How can we store a list of bindings of variables to types?
06-36: **Class Types**

- For built-in types, we did not need to store any extra information.
- For Class types, what extra information do we need to store?
  - The name and type of each instance variable
  - How can we store a list of bindings of variables to types?
  - As an environment!

06-37: **Class Types**

class simpleClass {
    int x;
    int y;
    boolean z;
}

void main() {
    simpleClass a;
    simpleClass b;
    int c;
    int d;

    a = new simpleClass();
    a.x = c;
}

06-38: **Class Types**

```
Type Environment:

Variable Environment

KeyStack
```

06-39: **Array Types**

- For arrays, what extra information do we need to store?
06-40: **Array Types**

- For arrays, what extra information do we need to store?
  - The base type of the array
  - For statically declared arrays, we might also want to store range of indices, to add range checking for arrays
    - Will add some run time inefficiency – need to add code to dynamically check each array access to ensure that it is within the correct bounds
    - Large number of attacks are based on buffer overflows

06-41: **Array Types**

- Much like built-in types, we want only one instance of the internal representation for int[], one representation for int[][], and so on
  - So we can do a simple pointer comparison to determine if types are equal
  - Otherwise, we would need to parse an entire type structure whenever a type comparison needed to be done (and type comparisons need to be done frequently in semantic analysis!)

06-42: **Array Types**

```c
void main () {
    int w;
    int x[];
    int y[];
    int z[][];

    /* Body of main program */
}
```

06-43: **Class Types**

![Class Types Diagram]

06-44: **Semantic Analysis Overview**
• A Semantic Analyzer traverses the Abstract Syntax Tree, and checks for semantic errors
  • When declarations are encountered, proper values are added to the correct environment

06-45: Semantic Analysis Overview
• A Semantic Analyzer traverses the Abstract Syntax Tree, and checks for semantic errors
  • When a statement is encountered (such as \( x = 3 \)), the statement is checked for errors using the current environment
    • Is the variable \( x \) declared in the current scope?
    • Is it \( x \) of type \texttt{int}?

06-46: Semantic Analysis Overview
• A Semantic Analyzer traverses the Abstract Syntax Tree, and checks for semantic errors
  • When a statement is encountered (such as if \( x > 3 \) \( x++; \)), the statement is checked for errors using the current environment
    • Is the expression \( x > 3 \) a valid expression (this will require a recursive analysis of the expression \( x > 3 \))?
    • Is the expression \( x > 3 \) of type \texttt{boolean}?
    • Is the statement \( x++; \) valid (this will require a recursive analysis of the statement \( x++; \))?

06-47: Semantic Analysis Overview
• A Semantic Analyzer traverses the Abstract Syntax Tree, and checks for semantic errors
  • When a function definition is encountered:
    • Begin a new scope
    • Add the parameters of the functions to the variable environment
    • Recursively check the body of the function
    • End the current scope (removing definitions of local variables and parameters from the current environment)

06-48: Variable Declarations
• \texttt{int} \( x; \)
  • Look up the type \texttt{int} in the type environment.
    • (if it does not exist, report an error)
  • Add the variable \( x \) to the current variable environment, with the type returned from the lookup of \texttt{int}

06-49: Variable Declarations
• \texttt{foo} \( x; \)
  • Look up the type \texttt{foo} in the type environment.
    • (if it does not exist, report an error)
  • Add the variable \( x \) to the current variable environment, with the type returned from the lookup of \texttt{foo}

06-50: Array Declarations
- int A[];
  - Defines a variable A
  - Also potentially defines a type int[]

06-51: **Array Declarations**

- int A[];
  - look up the type int[] in the type environment
  - If the type exists:
    - Add A to the variable environment, with the type returned from looking up int[]

06-52: **Array Declarations**

- int A[];
  - look up the type int[] in the type environment
  - If the type does not exist:
    - Check to see if int appears in the type environment. If it does not, report an error
    - If int does appear in the type environment
      - Create a new Array type (using the type returned from int as a base type)
      - Add new type to type environment, with key int[]
      - Add variable A to the variable environment, with this type

06-53: **Multidimensional Arrays**

- For multi-dimensional arrays, we may need to repeat the process
- For a declaration int x[][][], we may need to add:
  - int[]
  - int[][]
  - int[][][]

  to the type environment, before adding x to the variable environment with the type int[][][]

06-54: **Multidimensional Arrays**

```c
void main() {
    int A[]][[];
    int B[];
    int C[]][[];

    /* body of main */
}
```

- For A[][][]:
  - Add int[], int[][], int[][][] to type environment
  - Add A to variable environment with type int[][][]
Multidimensional Arrays

```c
void main() {
    int A[][[]];
    int B[];
    int C[][[]];

    /* body of main */
}
```

- For `B[]`:
  - `int[]` is already in the type environment.
  - add `B` to variable environment, with the type found for `int[]`

Multidimensional Arrays

```c
void main() {
    int A[][[]];
    int B[];
    int C[][[]];

    /* body of main */
}
```

- For `C[][[]]`:
  - `int[]` is already in the type environment
  - add `C` to variable environment with type found for `int[]`

Multidimensional Arrays

- For the declaration `int A[][[]]`, why add types `int[]`, `int[][]`, and `int[][][]` to the type environment?
- Why not just create a type `int[][][]`, and add `A` to the variable environment with this type?
- In short, why make sure that all instances of the type `int[]` point to the same instance? (examples)

Multidimensional Arrays

```c
void Sort(int Data[]);
```

```c
void main() {
    int A[];
    int B[];
    int C[][[]];

    /* Code to allocate space for A, B & C, and set initial values */

    Sort(A);
    Sort(B);
    Sort(C[2]);
}
```
06-59: **Function Prototypes**

- `int foo(int a, boolean b);`
- Add a description of this function to the function environment

06-60: **Function Prototypes**

- `int foo(int a, boolean b);`
- Add a description of this function to the function environment
  - Type of each parameter
  - Return type of the function

06-61: **Function Prototypes**

```
int foo(int a, boolean b);
```

06-62: **Function Prototypes**

- `int PrintBoard(int board[][]);`
- Analyze types of input parameter
  - Add `int[]` and `int[][]` to the type environment, if not already there.

06-63: **Class Definitions**

class MyClass {
    int integerval;
    int Array[];
    boolean boolval;
}

06-64: **Class Definitions**
class MyClass {
    int integerval;
    int Array[];
    boolean boolval;
}

- Create a new variable environment

06-65: Class Definitions

class MyClass {
    int integerval;
    int Array[];
    boolean boolval;
}

- Create a new variable environment
- Add integerval, Array, and boolval to this environment (possibly adding int[] to the type environment)

06-66: Class Definitions

class MyClass {
    int integerval;
    int Array[];
    boolean boolval;
}

- Create a new variable environment
- Add integerval, Array, and boolval to this environment (possibly adding int[] to the type environment)
- Add entry in type environment with key MyClass that stores the new variable environment

06-67: Function Prototypes

Type Environment:

06-68: Function Definitions
• Analyze formal parameters & return type. Check against prototype (if there is one), or add function entry to function environment (if no prototype)

• Begin a new scope in the variable environment

• Add formal parameters to the variable environment

• Analyze the body of the function, using modified variable environment

• End current scope in variable environment

06-69: **Expressions**

• To analyze an expression:
  • Make sure the expression is well formed (no semantic errors)
  • Return the type of the expression (to be used by the calling function)

06-70: **Expressions**

• Simple Expressions
  • 3 (integer literal)
    • This is a well formed expression, with the type int
  • true (boolean literal)
    • This is a well formed expression, with the type boolean

06-71: **Expressions**

• Operator Expressions
  • 3 + 4
    • Recursively find types of left and right operand
    • Make sure the operands have integer types
    • Return integer type
  • x ÷ 3
    • Recursively find types of left and right operand
    • Make sure the operands have integer types
    • Return boolean type

06-72: **Expressions**

• Operator Expressions
  • (x ÷ 3) —— z
    • Recursively find types of left and right operand
    • Make sure the operands have boolean types
    • Return boolean type

06-73: **Expressions – Variables**

• Simple (Base) Variables – x
• Look up \( x \) in the variable environment
• If the variable was in the variable environment, return the associated type.
• If the variable was not in the variable environment, display an error.
  • Need to return something if variable is not defined – return type integer for lack of something better

06-74: Expressions – Variables

• Array Variables – \( A[3] \)
  • Analyze the index, ensuring that it is of type int
  • Analyze the base variable. Ensure that the base variable is an Array Type
  • Return the type of an element of the array, extracted from the base type of the array

  ```
  int A[];
  /* initialize A, etc. */
  x = A[3];
  ```

06-75: Expressions – Variables

• Array Variables
  • Analyze the index, ensuring that it is of type int
  • Analyze the base variable. Ensure that the base variable is an Array Type
  • Return the type of an element of the array, extracted from the base type of the array

  ```
  int B[][];
  /* initialize B, etc. */
  x = B[3][4];
  ```

06-76: Expressions – Variables

• Array Variables
  • Analyze the index, ensuring that it is of type int
  • Analyze the base variable. Ensure that the base variable is an Array Type
  • Return the type of an element of the array, extracted from the base type of the array

  ```
  int B[][];
  int A[];
  /* initialize A, B, etc. */
  ```

06-77: Expressions – Variables

• Array Variables
  • Analyze the index, ensuring that it is of type int
  • Analyze the base variable. Ensure that the base variable is an Array Type
• Return the type of an element of the array, extracted from the base type of the array

```java
int B[];  
int A[];  
/* initialize A, B, etc. */  
```

06-78: **Expressions – Variables**

• **Instance Variables – x.y**
  
  • Analyze the base of the variable (x), and make sure it is a class variable.
  
  • Look up y in the variable environment *for the class x*
  
  • Return the type associated with y in the variable environment for the class x.

06-79: **Instance Variables**

```java
class foo {  
  int x;  
  boolean y;
}

int main() {  
  foo x;  
  int y;  
  ...  
  y = x.x;  
  y = x.y;
}
```

06-80: **Instance Variables**

```java
class foo {  
  int x;  
  boolean y[];
}

int main() {  
  foo A[];  
  int a;  
  boolean b;  
  ...  
  w = A[3].x;  
  b = A[3].y[4];  
}
```

06-81: **Statements**

• If statements
  
  • Analyze the test, ensure that it is of type boolean
- Analyze the “if” statement
- Analyze the “else” statement (if there is one)

06-82: Statements
- Assignment statements
  - Analyze the left-hand side of the assignment statement
  - Analyze the right-hand side of the assignment statement
  - Make sure the types are the same
    - Can do this with a simple pointer comparison!

06-83: Statements
- Block statements
  - Begin new scope in variable environment
  - Recursively analyze all children
  - End current scope in variable environment

06-84: Statements
- Variable Declaration Statements
  - Look up type of variable
    - May involve adding types to type environment for arrays
  - Add variable to variable environment
  - If there is an initialization expression, make sure the type of the expression matches the type of the variable.

06-85: Types in Java
- Each type will be represented by a class
- All types will be subclasses of the “type” class:

class Type {  
}

06-86: Built-in Types
- Only one internal representation of each built-in type
  - All references to INTEGER type will be a pointer to the same block of memory
- How can we achieve this in Java?
  - Singleton software design pattern

06-87: Singletons in Java
- Use a singleton when you want only one instantiation of a class
Every call to “new” creates a new instance

– prohibit calls to “new”!
  • Make the constructor private
  • Obtain instances through a static method

06-88: **Singletons in Java**

```java
public class IntegerType extends Type {
    private IntegerType() {
    }

    public static IntegerType instance() {
        if (instance_ == null) {
            instance_ = new IntegerType();
        }
        return instance_;
    }

    static private IntegerType instance_;
}
```

06-89: **Singletons in Java**

```java
Type t1;
Type t2;
Type t3;

    t1 = IntegerType.instance();
    t2 = IntegerType.instance();
    t3 = IntegerType.instance();
```

• t1, t2, and t3 all point to the same instance

06-90: **Structured Types in Java**

• Built-in types (integer, boolean, void) do not need any extra information
  • An integer is an integer is an integer

• Structured types (Arrays, classes) need more information
  • An array of what
  • What fields does the class have

06-91: **Array Types in Java**

• Internal representation of array type needs to store the element type of the array

```java
class ArrayType extends Type {
    public ArrayType(Type type) {
        type_ = type;
    }

    public Type type() {
        return type_;
    }

    public void settype(Type type) {
        type_ = type;
    }

    private Type type_;
}
```
06-92: **Array Types in Java**

- Creating the internal representation of an array of integers:
  
  \[
  \text{Type } t_1; \\
  t_1 = \text{new ArrayType(IntegerType.instance());}
  \]

- Creating the internal representation of a 2D array of integers:
  
  \[
  \text{Type } t_2; \\
  t_2 = \text{new ArrayType(new ArrayType(IntegerType.instance()));}
  \]

06-93: **Array Types in Java**

- Creating the internal representation of a 2D array of integers:
  
  \[
  \text{Type } t_2; \\
  t_2 = \text{new ArrayType(new ArrayType(IntegerType.instance()));}
  \]

- Note that you should not use this exact code in your semantic analyzer
  
  - Create a 1D array of integers, add this to the type environment
  - Create an array of 1D array of integers, using the previously created type

06-94: **Environments**

- TypeEnvironment.java
- TypeEntry.java
- VariableEnvironment.java
- VariableEntry.java
- FunctionEnvironment.java
- FunctionEntry.java

06-95: **Class Types**

- Create the type for the class:
  
  ```java
  class foo {
      int x;
      boolean y;
  }
  ```

- with the Java code:
  
  ```java
  Type t4; \\
  VariableEnvironment instanceVars = new VariableEnvironment(); \\
  instanceVars.insert("x", new VariableEntry(IntegerType.instance())); \\
  instanceVars.insert("y", new VariableEntry(BooleanType.instance())); \\
  t4 = new ClassType(instanceVars); 
  ```

06-96: **Reporting Errors**

- Class CompError:
public class CompError {
    private static int numberOfErrors = 0;
    public static void message(int linenum, String errstm) {
        numberOfErrors++;
        System.out.println("TstError in line " + linenum + ": " + errstm);
    }
    public static boolean anyErrors() {
        return numberOfErrors > 0;
    }
    public static int numberOfErrors() {
        return numberOfErrors;
    }
}

06-97: Reporting Errors

- Using CompError
- Trying to add booleans on line 12 ...

CompError.message(12, "Arguments to + must be integers");

06-98: Traversing the AST

- Write a Visitor to do Semantic Analysis
  - Method for each type of AST node
  - VisitProgram analyzes ASTProgram
  - VisitIfStatement analyzes an ASTStatement
  - ... etc.

06-99: Setting up the Visitor

public class SemanticAnalyzer implements ASTVisitor {
    private VariableEnvironment variableEnv;
    private FunctionEnvironment functionEnv;
    private TypeEnvironment typeEnv;
    /* May need to add some more ... */

    public SemanticAnalyzer() {
        variableEnv = new VariableEnvironment();
        functionEnv = new FunctionEnvironment();
        functionEnv.addBuiltinFunctions();
        typeEnv = new TypeEnvironment();
    }
}

06-100: Traversing the AST

public Object VisitProgram(ASTProgram program) {
    program.classes().Accept(this);
    program.functiondefinitions().Accept(this);
    return null;
}

06-101: Analyzing Expressions

- Visitor methods for expressions will return a type
  - Type of the expression that was analyzed
  - The return value will be used to do typechecking “upstream”

06-102: Analyzing Expressions
06-103: Analyzing Variables

- Three different types of variables
  - (Base, Array, Class)

```java
ASTVariable a, b, c;
ASTIntegerLiteral literal;  
Type t;
a = new ASTBaseVariable("x");
b = new ASTArrayVariable(a, new ASTIntegerLiteral(3));
c = new ASTClassVariable(b, "y");
t = (Type) a.Accept(semanticAnalyzer);
t = (Type) b.Accept(semanticAnalyzer);
t = (Type) c.Accept(semanticAnalyzer);
```

06-104: Base Variables

- To analyze a base variable
  - Look up the name of the base variable in the variable environment
  - Output an error if the variable is not defined
  - Return the type of the variable
    - (return something if the variable not declared. An integer is as good as anything.

```java
public Object VisitBaseVariable(ASTBaseVariable base) {
    VariableEntry baseEntry = variableEnv.find(base.name());
    if (baseEntry == null) {
        CompError.message(base.line(), "Variable " + base.name() + " is not defined in this scope");
        return IntegerType.instance();
    } else {
        return baseEntry.type();
    }
}
```

06-105: Analyzing Statements

- To analyze a statement
  - Recursively analyze the pieces of the statement
  - Check for any semantic errors in the statement
  - Don’t need to return anything (yet!) – if the statement is correct, don’t call the Error function!

06-107: Analyzing If Statements

- To analyze an if statement we:

```java
public Object VisitIfStatement(ASTIfStatement ifStatement) {
    Object thenResult = visit(ifStatement.thenStatement);
    Object elseResult = visit(ifStatement.elseStatement);
    Type testType = (Type) ifStatement.test.Accept(semanticAnalyzer);
    if (testType == null || !testType.isBoolean()) {
        CompError.message(ifStatement.line(), "If statement requires a boolean test");
        return null;
    }
   ...
```
public Object VisitIfStatement(ASTIfStatement ifsmt) {
    Type test = (Type) ifsmt.test().Accept(this);
    if (test != BooleanType.instance()) {
        CompError.message(ifsmt.line(), "If test must be a boolean");
    }
    ifsmt.thenstatement().Accept(this);
    if (ifsmt.elsestatement() != null) {
        ifsmt.elsestatement().Accept(this);
    }
    return null;
}

06-110: **Project Hints**

- This project will take *much* longer than the previous projects. You have 3 weeks (plus Spring Break) – start **NOW**.
- The project is pointer intensive. Spend some time to understand environments and type representations before you start.
- Start early. This project is longer than the previous three projects.
- Variable accesses can be tricky. Read the section in the class notes closely before you start coding variable analyzer.
- Start early. (Do you notice a theme here? I’m not kidding. Really.)