

Compilers

CS414-2003S-07

Semantic Analysis

David Galles

Department of Computer Science
University of San Francisco

07-0: 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:

```
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?

07-1: 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:
 - $L = \{ww \mid w \in \{a, b\}^*\}$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.

07-2: **yacc & CFGs**

- Yacc allows actions – arbitrary C code – in rules
- We could use yacc rules to do type checking
- Why don't we?

07-3: **yacc & CFGs**

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

07-4: Semantic Errors/Syntax Errors

- Thus, we only build the Abstract Syntax Tree in yacc (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.

07-5: 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

07-6: 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

07-7: Semantic Errors

- Semantic Errors can be classified into the following broad categories:
 - **Function and Method Errors**
 - `foo(3, true, 8)`
 - `foo` must be a function which takes 3 parameters:
 - integer
 - boolean
 - integer

07-8: 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, || &&, ! take booleans
 - Standard Java has polymorphic functions & type coercion

07-9: Semantic Errors

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

07-10: 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

07-11: 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;
```

07-12: 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 superclass value to a subclass variable

07-13: 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)

07-14: 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.

07-15: Environments & Name Spaces

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

simpleJava:

```
class foo {  
    int foo;  
}
```

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

07-16: Environments & Name Spaces

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

C:

```
#include <stdio.h>

typedef int foo;
int main() {
    foo foo;
    foo = 4;
    printf("%d", foo);
    return 0;
}
```

07-17: Environments & Name Spaces

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

Java:

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

07-18: Environments & Name Spaces

- Variables and functions in C share the same name space, so the following C code is not legal:

```
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`

07-19: 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 ...

07-20: 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.

07-21: Changing Environments

```
int foo(int x) {  
    boolean y;  
  
    x = 2;  
    y = false;  
    /* Position A */  
    { int y;  
      boolean z;  
  
      y = 3;  
      z = true;  
    } /* Position B */  
    /* Position C */  
}
```

07-22: 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

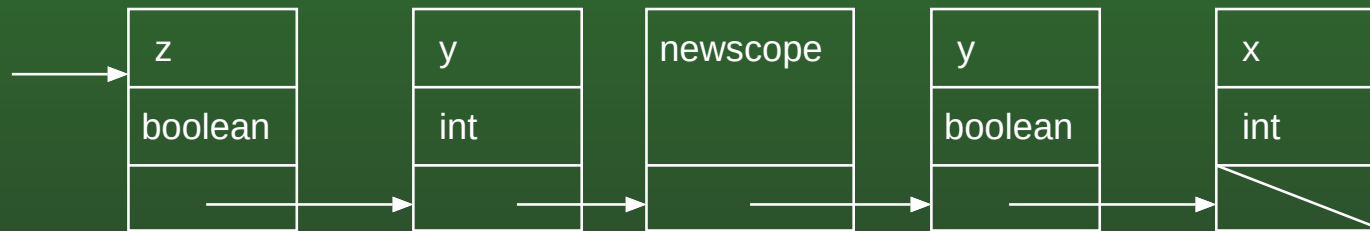
07-23: Implementing Symbol Tables

- 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

07-24: Implementing Symbol Tables



a



b



c

07-25: 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 = $\Theta(\text{\# of key/value pairs})$
- Then inserting and finding take time $\Theta(1)$

07-26: Hash Functions

```
long hash(char *key, int tableSize) {  
    long h = 0;  
    long g;  
    for (;*key;key++) {  
        h = (h << 4) + *key;  
        g = h & 0xF0000000;  
        if (g) h ^= g >> 24  
        h &= g  
    }  
    return h % tableSize;  
}
```

07-27: 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`?

07-28: 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`?

07-29: 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

07-30: 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)

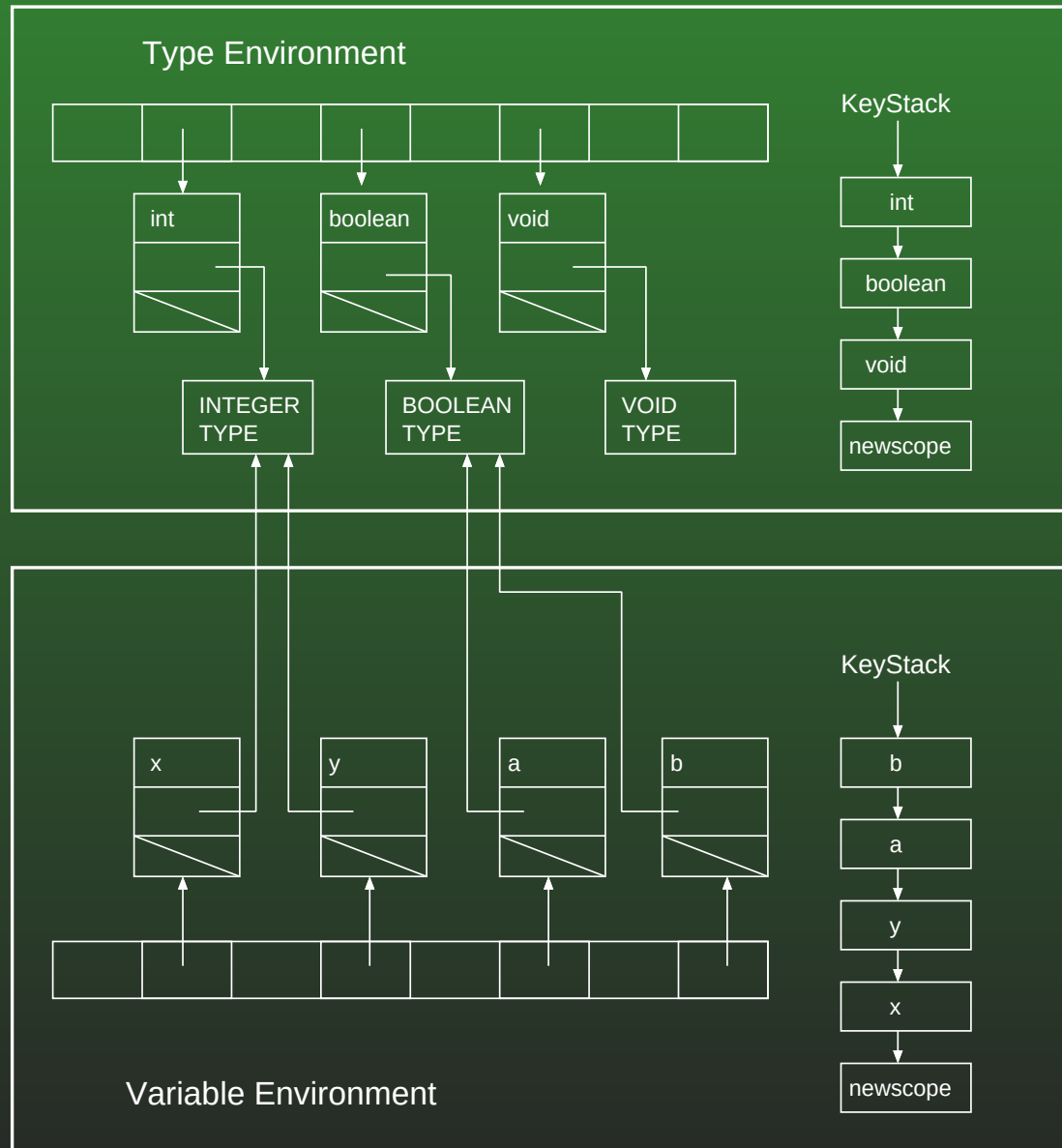
07-31: Built-In Types

- 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

07-32: Built-In Types

```
void main() {  
    int x;  
    int y;  
    boolean a;  
    boolean b;  
  
    x = y;  
    x = a;    /* Type Error */  
}
```

07-33: Built-In Types



07-34: **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?

07-35: **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?

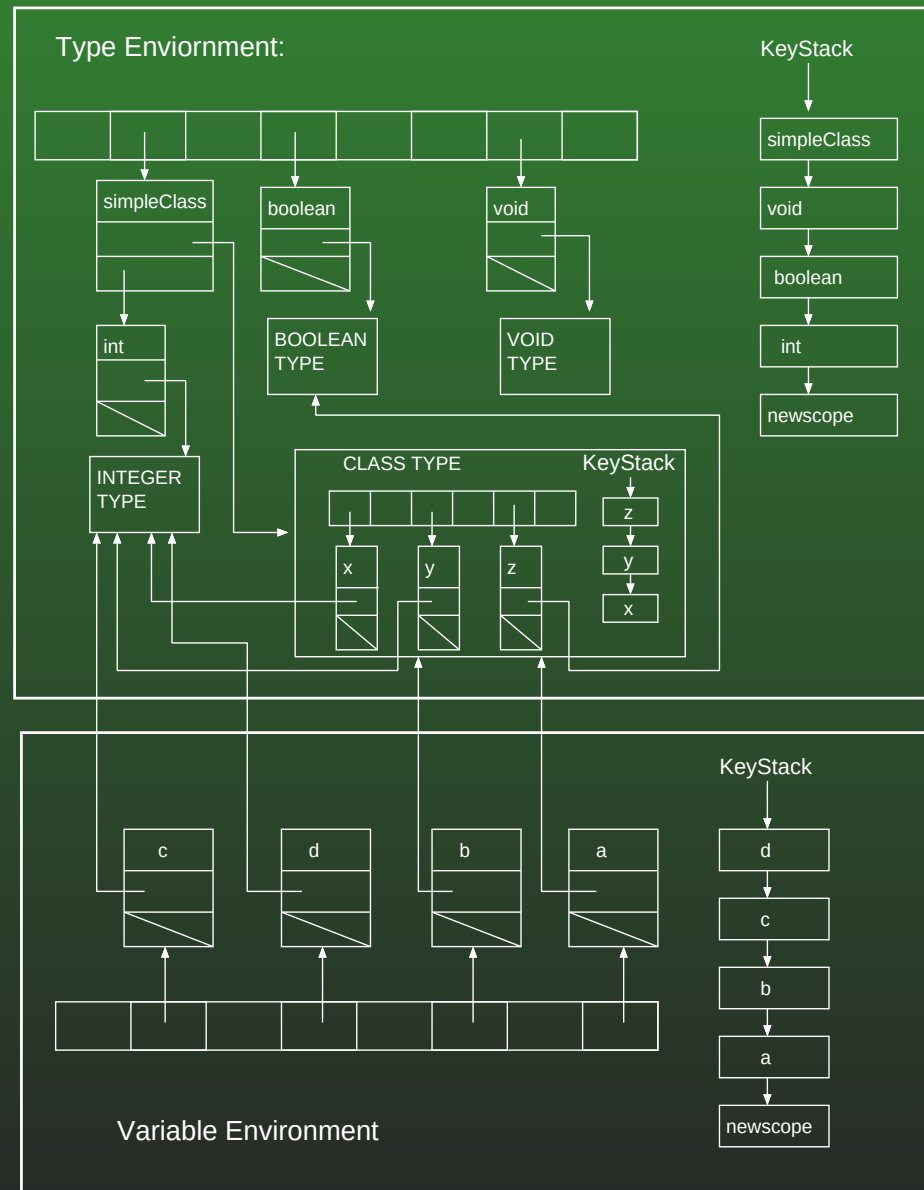
07-36: Class Types

- 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!

07-37: 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;  
}
```

07-38: Class Types



07-39: **Array Types**

- For arrays, what extra information do we need to store?

07-40: 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

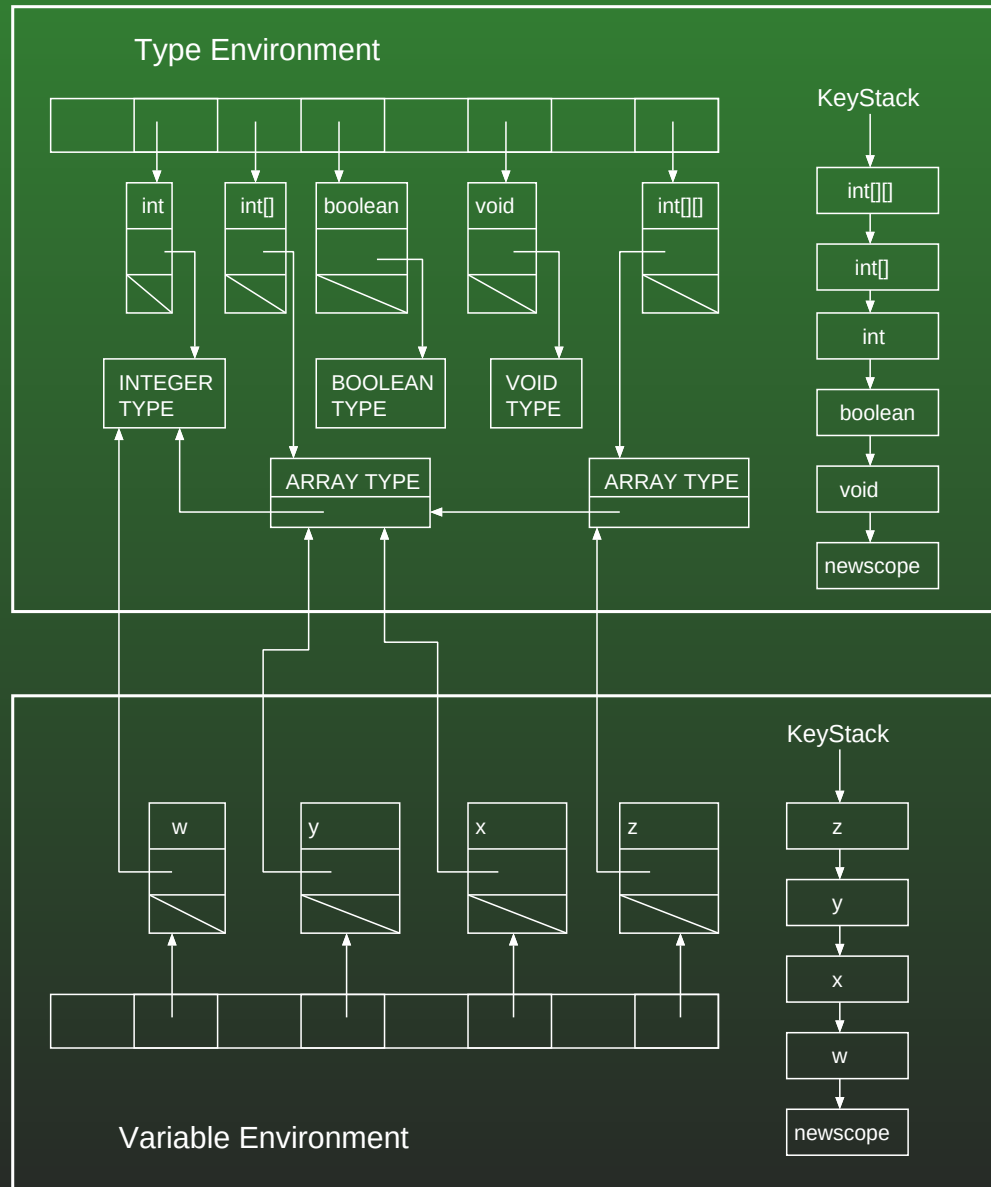
07-41: Array Types

- 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!)

07-42: Array Types

```
void main () {  
    int w;  
    int x[];  
    int y[];  
    int z[][];  
  
    /* Body of main program */  
}
```

07-43: Class Types



07-44: 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

07-45: 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 `int`?

07-46: 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 `boolean`?
 - Is the statement `x++` valid (this will require a recursive analysis of the statement `x++`;

07-47: 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)

07-48: Variable Declarations

- `int x;`
 - Look up the type `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 `int`

07-49: Variable Declarations

- `foo x;`
 - Look up the type `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 `foo`

07-50: **Array Declarations**

- `int A[];`
 - Defines a variable A
 - *Also* potentially defines a type `int[]`

07-51: 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[]`

07-52: 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

07-53: 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[][][]`

07-54: Multidimensional Arrays

```
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[][][]

07-55: Multidimensional Arrays

```
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[]

07-56: Multidimensional Arrays

```
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[][]

07-57: 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)

07-58: Multidimensional Arrays

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

```
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]);
```

```
}
```

07-59: **Function Prototypes**

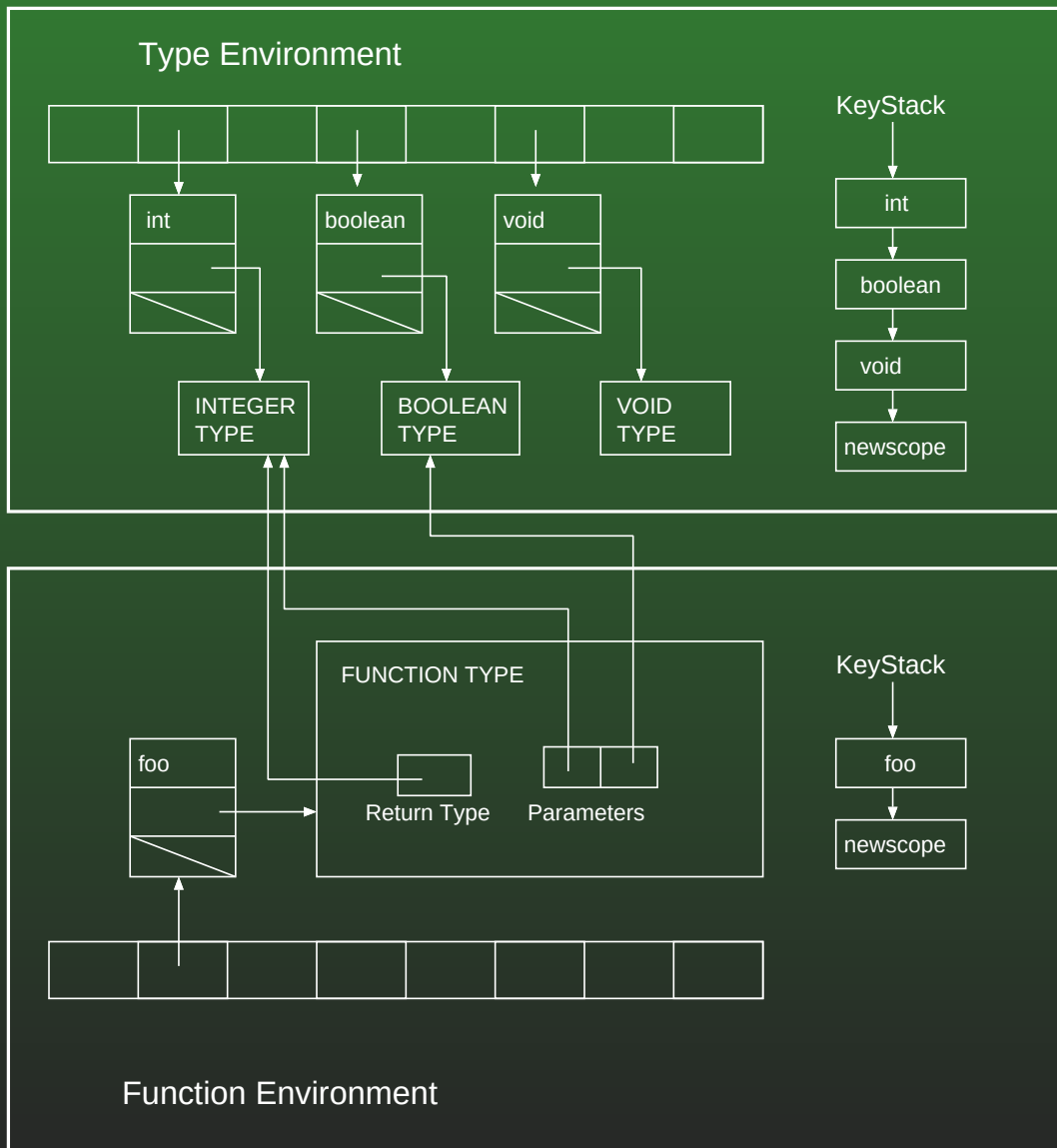
- `int foo(int a, boolean b);`
- Add a description of this function to the function environment

07-60: **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

07-61: Function Prototypes

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



07-62: **Function Prototypes**

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

07-63: Class Definitions

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

07-64: Class Definitions

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

- Create a new variable environment

07-65: 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)

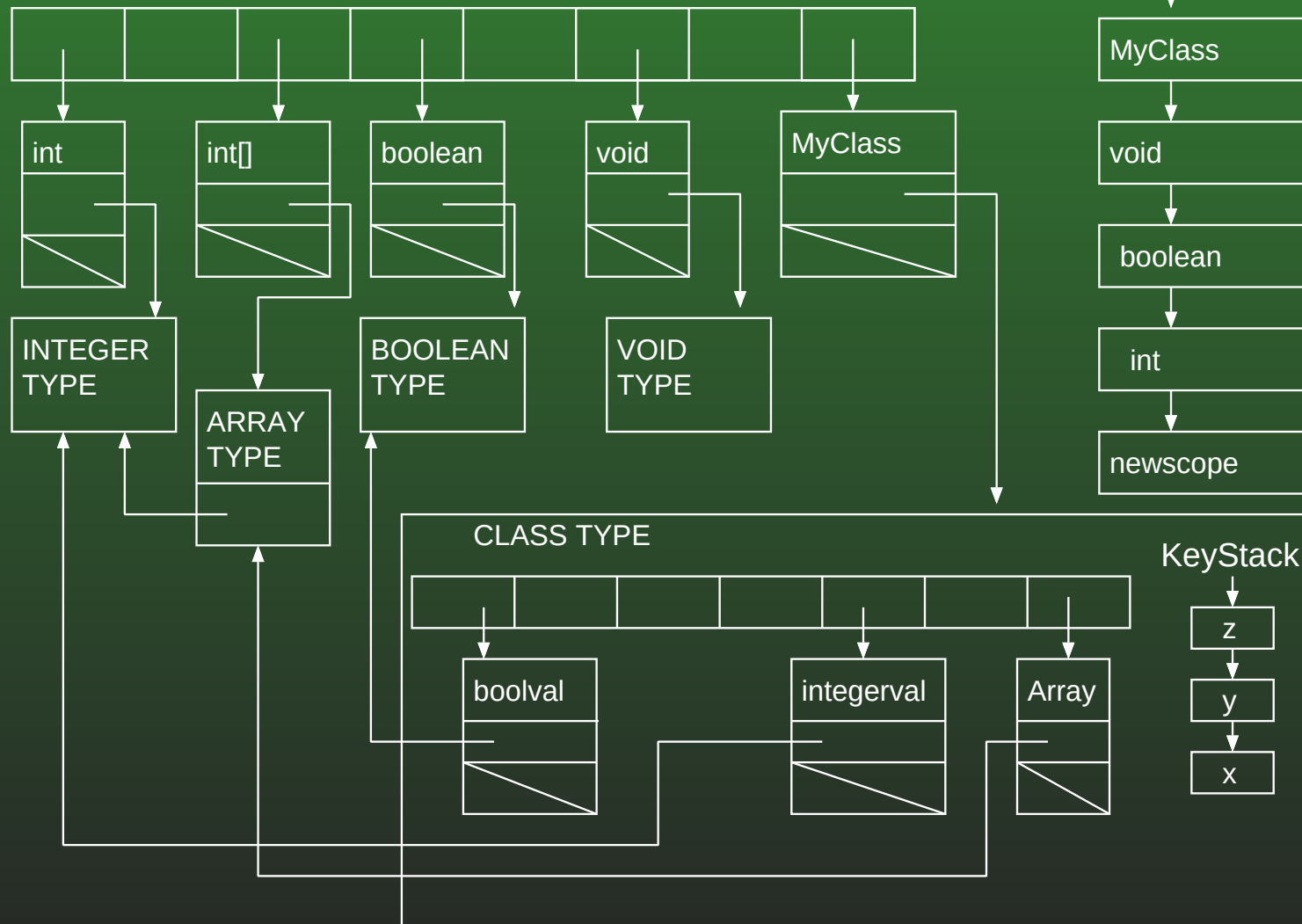
07-66: Class Definitions

```
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

07-67: Function Prototypes

Type Environment:



07-68: **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

07-69: 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)

07-70: 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 int

07-71: 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

07-72: Expressions

- Operator Expressions
 - $(x > 3) \parallel z$
 - Recursively find types of left and right operand
 - Make sure the operands have boolean types
 - Return boolean type

07-73: 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

07-74: 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];
```

07-75: 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] ;
```

07-76: 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. */
x = B[A[4]] [A[3]] ;
```

# 07-77: 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

- ```
int B[] [] ;  
int A[] ;
```

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

07-78: 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 .

07-79: Instance Variables

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

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

07-80: Instance Variables

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

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

07-81: 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)

07-82: 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!

07-83: Statements

- Block statements
 - Begin new scope in variable environment
 - Recursively analyze all children
 - End current scope in variable environment

07-84: 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.

07-85: Types in C

- Three different kinds of types:
 - Builtins (int, boolean, void)
 - Array
 - Base type
 - Class
 - Name (and types) of all instance variables
 - – *environment*

07-86: **Types in C**

```
typedef struct type_ *type;

struct type_ {
    enum {integer_type, boolean_type, void_type,
          class_type, array_type} kind;
    union {
        type array;
        struct {
            environment instancevars;
        } class;
    } u;
};
```


07-87: Built-in Types

- *One* instance of each of the base types
- Each call to constructor to return the *same* instance:

```
type t1,t2;  
t1 = IntegerType();  
t2 = IntegerType();  
if (t1 == t2) {  
    ...  
}
```

- if test should always be true

07-88: Built-in Types

```
type integerType_ = NULL;

type IntegerType() {
    if (integerType_ == NULL) {
        integerType_ = (type) malloc(sizeof(type_));
        integerType_->kind = integer_type;
    }
    return integerType_;
}
```

07-89: **Array Types**

- `int A[];`
 - Create the type with `ArrayType(IntegerType());`
- `int A[][];`
 - Create the type with `ArrayType(ArrayType(IntegerType()));`

07-90: Environments

- File environment1.h

```
typedef struct environment_ *environment;
typedef struct envEntry_ *envEntry;

environment Environment();
void AddBuiltinTypes(environment env);
void AddBuiltinFunctions(environment env);

void beginScope(environment env);
void endScope(environment env);
void enter(environment env, char * key, envEntry entry);
envEntry find(environment env, char *key);
```

07-91: Environments

- File environment2.h

```
envEntry VarEntry(type typ);
envEntry FunctionEntry(type returntyp, typeList formals);
envEntry TypeEntry(type typ);
struct envEntry_ {
    enum {Var_Entry, Function_Entry, Type_Entry} kind;
    union {
        struct {
            type typ;
        } varEntry;
        struct {
            type returntyp;
            typeList formals;
        } functionEntry;
        struct {
            type typ;
        } typeEntry;
    } u;
};
```

07-92: Class Types

- Create the type for the class:

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

- with the C code:

```
type t4;  
environment instanceVars = Environment();  
enter(instanceVars, "x",  
      VariableEntry(IntegerType()));  
enter(instanceVars, "y",  
      VariableEntry(BooleanType()));  
t4 = ClassType(instanceVars);
```

07-93: Reporting Errors

- Function Error:

```
char *name;  
int intval;  
  
Error(3, "Variable %s not defined", name);  
Error(15, "Function %s requires %d input parmeters",  
      name, intval);
```

07-94: Reporting Errors

- File errors.h

```
void Error(int linenum, char *message, ...);  
int anyerrors();  
int numerrors();
```


07-95: Traversing the AST

- Write a suite of functions to traverse the tree
 - Function for each type of tree node
 - function for ASTprogram analyzes an ASTprogram
 - function for ASTstatement analyzes an ASTstatement
 - ... etc.

07-96: Traversing the AST

```
void analyzeProgram(ASTprogram program) {
    environment typeEnv;
    environment functionEnv;
    environment varEnv;

    typeEnv = Environment();
    functionEnv = Environment();
    varEnv = Environment();

    AddBuiltinTypes(typeEnv);
    AddBuiltinFunctions(functionEnv);

    analyzeClassList(typeEnv, functionEnv, varEnv, program->classes);
    analyzeFunctionDecList(typeEnv, functionEnv, varEnv, program->functiondecs);
}
```

07-97: Analyzing Expressions

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

07-98: Analyzing Expressions

```
void analyzeExpression(typeEnvironment typeEnv,
                      functionEnvironment functionEnv,
                      variableEnvironment varEnv, ASTexpression expression) {
    switch(expression->kind) {
    case ASTvariableExp:
        return analyzeVariable(varEnv, exp->u.var);
    case ASTintegerLiteralExp:
        return IntegerType();
    case ASTbooleanLiteral:
        return BooleanType();
    case ASToperatorExp:
        return analyzeOperatorExpression(typeEnv, functionEnv, varEnv, expression);
    case ASTcallExp:
        return analyzeCallExpression(typeEnv, functionEnv, varEnv, expression);
    }
}
```

07-99: Analyzing Variables

- Three different types of variables
 - (Base, Array, Class)
- Examine the “kind” field to determine which kind
- Call appropriate function

07-100: 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.

07-101: Base Variables

```
type analyzeBaseVariable(variableEnvironment varEnv, ASTVariable var) {
    envEntry base;
    base = find(varEnv, var->u.baseVar.name);
    if (base == NULL) {
        Error(var->line, "Variable %s not defined", var->u.baseVar.name);
        return IntegerType();
    }
    return base->u.typeEntry.typ;
}
```

07-102: 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!

07-103: Analyzing Statements

```
void analyzeStatement(environment typeEnv, environment functionEnv,
    environment varEnv, ASTstatement statement) {

    switch(statement->kind) {
    case AssignStm:
        analyzeAssignStm(typeEnv, functionEnv, varEnv, statement);
        break;
    case IfStm:
        analyzeIfStm(typeEnv, functionEnv, varEnv, statement);
        break;

    /* lots of other statements */

    case EmptyStm:
        break;
    default:
        Error(statement->line, "Bad Statement");
    }
}
```

07-104: **Analyzing If Statements**

- To analyze an if statement we:

07-105: **Analyzing If Statements**

- To analyze an if statement we:
 - Recursively analyze the “then” statement (and the “else statement, if it exists)
 - Analyze the test
 - Make sure the test is of type boolean

07-106: Analyzing If Statements

```
void analyzeIfStm(environment typeEnv, environment varEnv,  
                  environment functionEnv, ASTstatement statement) {  
    type testType;  
    type test = analyzeExpression(typeEnv, functionEnv, varEnv,  
                                  statement->u.ifStm.test);  
    if (test != BooleanType()) {  
        Error(statement->line, "If test must be a boolean");  
    }  
    analyzeStatement(typeEnv, functionEnv, varEnv, statement->u.ifStm.thenstm);  
    analyzeStatement(typeEnv, functionEnv, varEnv, statement->u.ifStm.elsestm);  
}
```

07-107: 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.)