**MAHARISHI UNIVERSITY OF MANAGEMENT**

# COMPUTER SCIENCE DEPARTMENT

# COMP 440 DE – Compilers

## Symbol Table Classes

In this programming assignment, you are to write Java classes to implement a symbol table. The purpose of the *symbol table* is to record bindings and manage the scope of these bindings. Bindings are created by declarations and have a specific scope. A *binding* is an association between an identifier name and a program entity, i.e., it is a mapping of an identifier to a specific program entity. For example, entities, such as variables and methods, can be named in a declaration and then referenced in other parts of the program using that name. This is handled in a compiler by recording a binding in the symbol table whenever a declaration is encountered. In almost all modern programming languages, the scope of a binding or declaration is determined by the syntactic structure of the program. The *scope* of a binding is the portion of program text where that binding is visible. The kinds of scope we will be dealing with in this and later lab assignments are global scope, class scope, method scope, and block scope. In this programming assignment you will define a symbol table as well as Java classes for each kind of scope and entity (for later use in our compiler project).

---

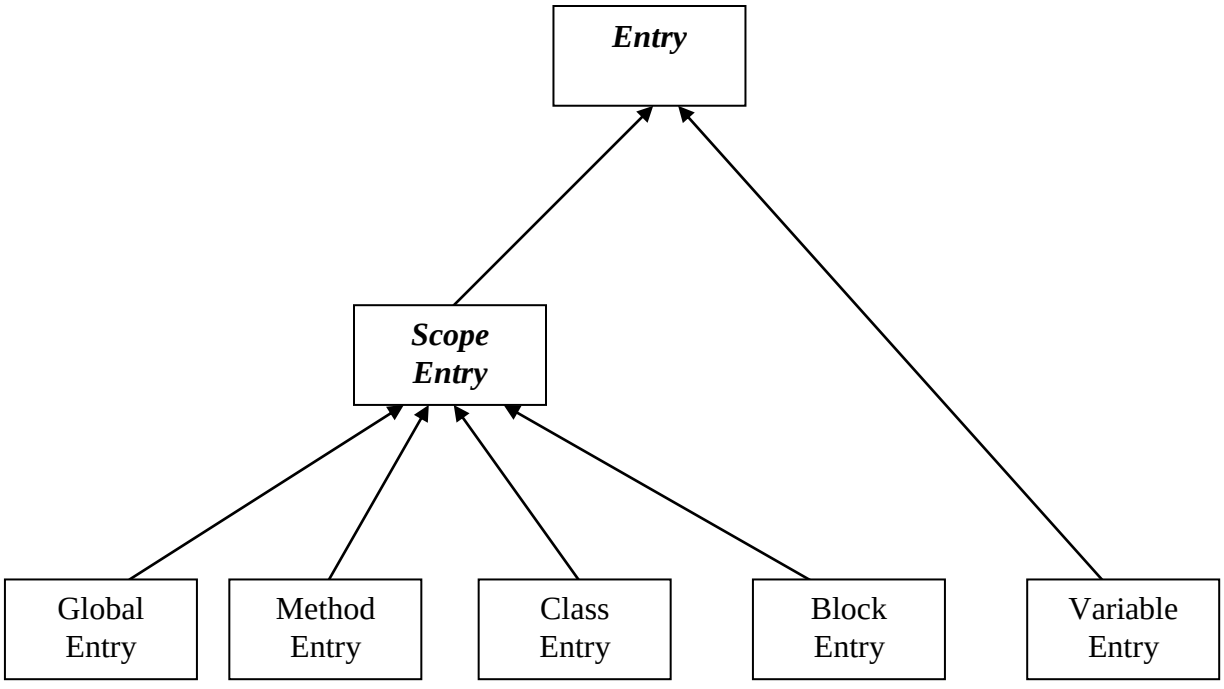


**Figure 1:** Class hierarchy of Symbol Table classes.

Our compiler will be for a programming language that we will refer to as C-plus. Our language will be called C-plus because it will be a small subset of C but extended to include classes as used in the C++ programming language

(instead of struct records). A class allows a programmer to group related declarations in separate modules to prevent name conflicts between declarations in different classes. However, classes, as in C++, may contain (static) method and field declarations, i.e., the use of a class is similar to declaring a Java class containing only static members (e.g., static fields and static methods).

**Program entities**

The symbol table must contain information about each program entity, i.e., properties such as the type of a variable or the signature of a method. The hierarchy of classes used to hold information about the various entities is given in **Figure 1**. Each entry class represents a program entity and contains the properties of that particular entity. **Figure 1** shows the various entry classes that have to be defined in this assignment. The information contained in the various entities is as follows:

- A variable entry (for a field or parameter) contains its name and type.
- A method entry contains its name, return type, and bindings for its parameters.
- A block entry contains bindings for the variables declared within that block.
- A class entry contains its name and the bindings for the entities declared within it.
- The global entry object contains bindings for the entities declared in the global scope; there must be one and only one global entry in the Symbol Table.

**Nested scopes and nested entries**

Note also that, as in Java, C#, and C++, entities can be nested inside other entities. For example, global, class, method, and block entries can contain the bindings for other entities that were declared within them. When entries are nested, an inner entry will be stored as a binding so it can be retrieved using its name (an identifier). The inner entry objects will also have to be stored so they can be retrieved in the order they were declared in the program (see below and the comments in the skeleton programs for more implementation details).

The symbol table must provide a correct mapping between each identifier name and the entity it denotes. Therefore, the symbol table must distinguish between identifiers declared in different (sometimes nested) scopes. The nesting of entries is necessary to handle nested scopes and, in particular, when the same identifier name denotes more than one distinct entity. Consider, for example, the following C-plus program fragment:

```
class N {
    String v;
    float f;

    int w(char t) {
        int v;
    }

    void r(int x, int y) {
        char w;
    }
}
```

**Figure2:** A C-plus program fragment.

In **Figure 2**, the identifier **v** refers to two distinct entities, i.e. variables (one with type String and the other with type int). Furthermore, the identifier **w** refers to two different kinds of entities, i.e., the method **w** in class **N** and the local variable declared in the body of method **r**. The multiple bindings for identifiers **v** and **w** do not conflict because they are declared in different (but nested) scopes, i.e., in a class scope and in a block scope nested inside that class. When determining which entity an identifier denotes (i.e. when looking a name up in the symbol table), a declaration in an inner scope has precedence over a declaration for the same identifier name in an outer scope. That is, a binding for an identifier declared in an inner scope *hides* the declaration of the same identifier in an outer scope. For example, in the body of method **w**, local variable **v** (int) hides the outer declaration of **v** (String). To access the variable declared in the outer scope, we have to use a qualified name, i.e., **N.v**.

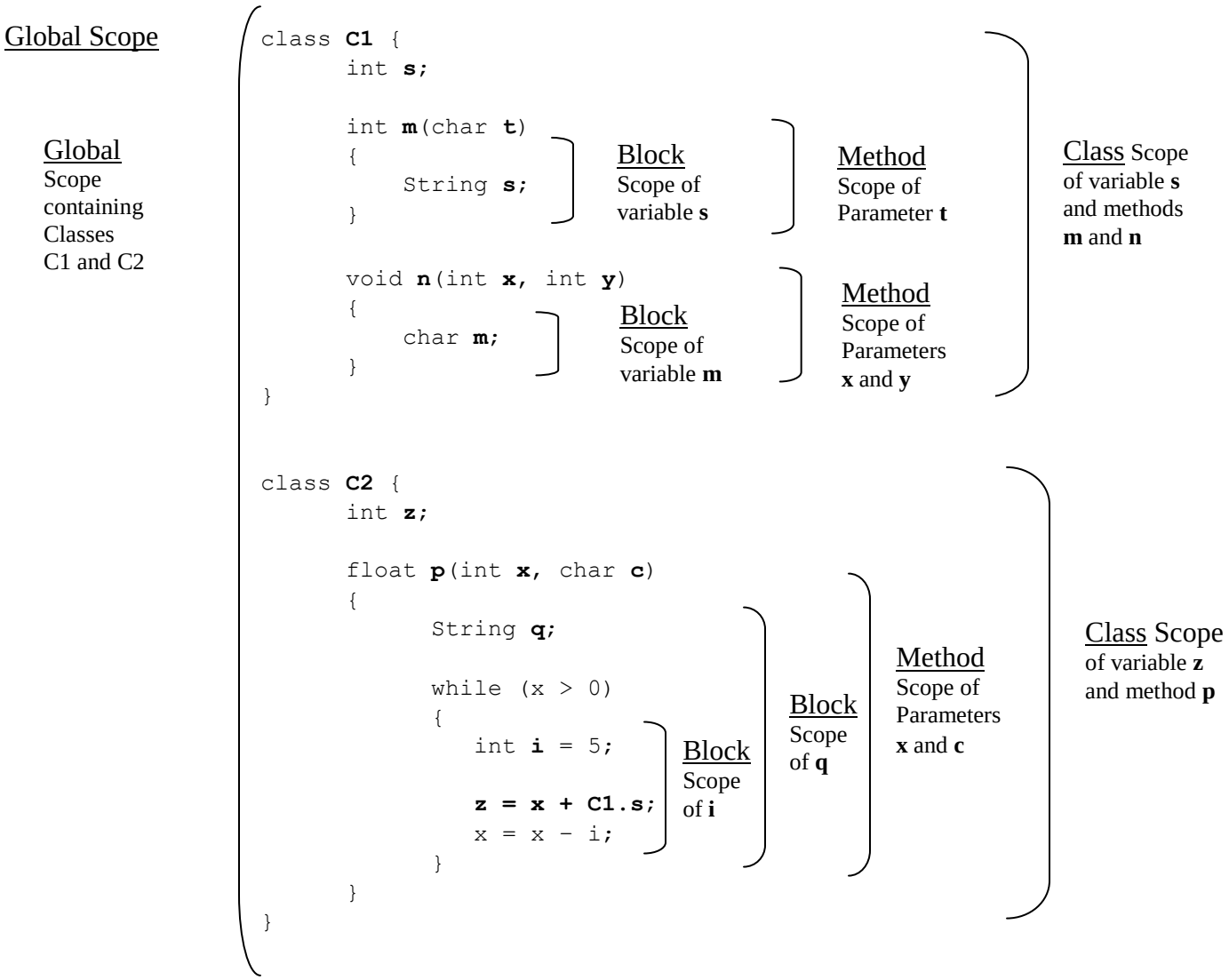


**Figure 3**: Diagram showing the nesting of scopes.

**Figure 3** illustrates the declarations (i.e. bindings) allowable within each kind of scope. For example

- Class definitions can occur within the global scope (e.g., classes **C1** and **C2**).
- Method declarations can occur within class scope (e.g., methods **m** and **n** in class **C1**).
- Variable declarations can occur within both class scope and block scope (e.g., variables **z**, **q**, and **i** in class **C2**).
- Parameter declarations can only occur in method scope (e.g., parameters **x** and **c** in method **p**).

**Figure 3** shows how the scopes of declarations can be nested. As already mentioned, the way we will represent and manage this nesting of scopes is to nest entries within other entries; this works as long as our symbol table and its bindings have the same nesting structure as the program itself. In the example in **Figures 3** and **4**, global bindings for classes **C1** and **C2** would be recorded inside the global scope entry. Therefore, a mapping from **C2** to its *ClassEntry* object would be recorded inside a *GlobalEntry* object. Similarly, bindings for **z** and **p**, declared inside class **C2**, would be recorded in the *ClassEntry* object representing **C2**. (Although not shown in **Figure 4**, bindings for **s**, **m**, and **n**, declared inside class **C1**, would also be recorded in the *ClassEntry* object representing class **C1**.) Furthermore, bindings for local variables like **q** and **i** in the blocks inside method **p** would be recorded in *BlockEntry* objects; and finally, parameter bindings would be recorded inside a *MethodEntry* object representing the method where the parameters are declared, e.g., **x** and **c** in the *MethodEntry* representing **p**.

As illustrated by **Figure 3**, there can be several levels of nested scopes. Therefore, when looking for a binding, the symbol table will have to search the current scope entry and, as necessary, search each enclosing scope entry until it finds the correct binding. For example, when inside the body of a method, the symbol table will have to first look for local variable bindings in one or more blocks; it must next look in the method scope for parameter bindings, then in the class scope, and finally in the global scope. Therefore, the symbol table will have to have the bindings from several scope entries available at the same time. **Figure 4** shows how nested scopes will be represented in the symbol table. In particular, **Figure 4** shows what the *SymbolTable* looks like when processing the following statement inside the while loop of method **p**:

```
z = x + C1.s;
```

When looking up variable **x** in the above statement, the symbol table (as illustrated in **Figure 4**) first looks in the *BlockEntry* representing the current block scope (the *BlockEntry* containing **i**). Since **x** is not there, the symbol table next looks in the surrounding block scope containing **q** (i.e., in the next scope entry). Since **x** is also not there, the symbol table next looks in the *MethodEntry* representing the scope of method **p**. Since **x** is one of **p**'s parameters, the symbol table returns the *VariableEntry* object representing the parameter **x**, i.e., the variable found when **x** is looked up in **p**'s *MethodEntry*.

When looking up a qualified name such as **C1.s**, we know that **C1** denotes a class. Therefore, the symbol table must first find **C1** by looking for **C1** in each surrounding scope until it is found. If a *ClassEntry* object is not found, then null is returned. However, in this case, the *ClassEntry* representing **C1** would be found; we must next look up **s** in the *ClassEntry* object representing **C1**. In this case, the symbol table returns the *VariableEntry* representing **s** (since it was declared in **C1**). More of the implementation details are given in the comments located in the skeleton files provided.

To build a symbol table that can be used in type checking, bindings must be recorded in the correct *ScopeEntry* object, i.e., the one representing the scope where the declaration is located since each *ScopeEntry* represents the context where a binding is visible. Therefore, when adding the binding created by a declaration, the entry representing the scope where that declaration is located must be the "current scope", i.e., the one on top of the scope stack (see example in **Figure 4**). In this way the symbol table will be used to manage scope entries so bindings will be added to the correct scope entry.

More specifically, scope entries will have to be managed like the run-time stack is managed during program execution, i.e., a scope entry has to be pushed onto the symbol table's scope stack each time a new context is entered, i.e., each time a class, method, or block scope is entered. Thus the symbol table must be able to enter and exit scopes as they are encountered. To enter a new scope, the *ScopeEntry* object representing the new scope must become the *current scope* (the entry on top of the scope stack in the symbol table), i.e., it must be pushed onto the stack of scope entries. The symbol table then leaves that scope by popping the current scope entry off that stack of scope entries.

Each time a declaration is encountered in a CP program, a new entry object is created and a binding for it is added to the current scope entry (the one on top of the stack). After traversing the program and building the symbol table, all the necessary information will have been recorded in the global entry object of the symbol table and all bindings will have been recorded in the appropriate scope entries in the global context; thus all class, method, and variable entries will have been recorded in their proper context (scope entry) object.

After a symbol table is built, it can be used to type check the program it represents (i.e., programs with nested scopes and forward references can be type checked); in particular, Lab 4 will build the symbol table and Lab 5 will type check the program using that symbol table.

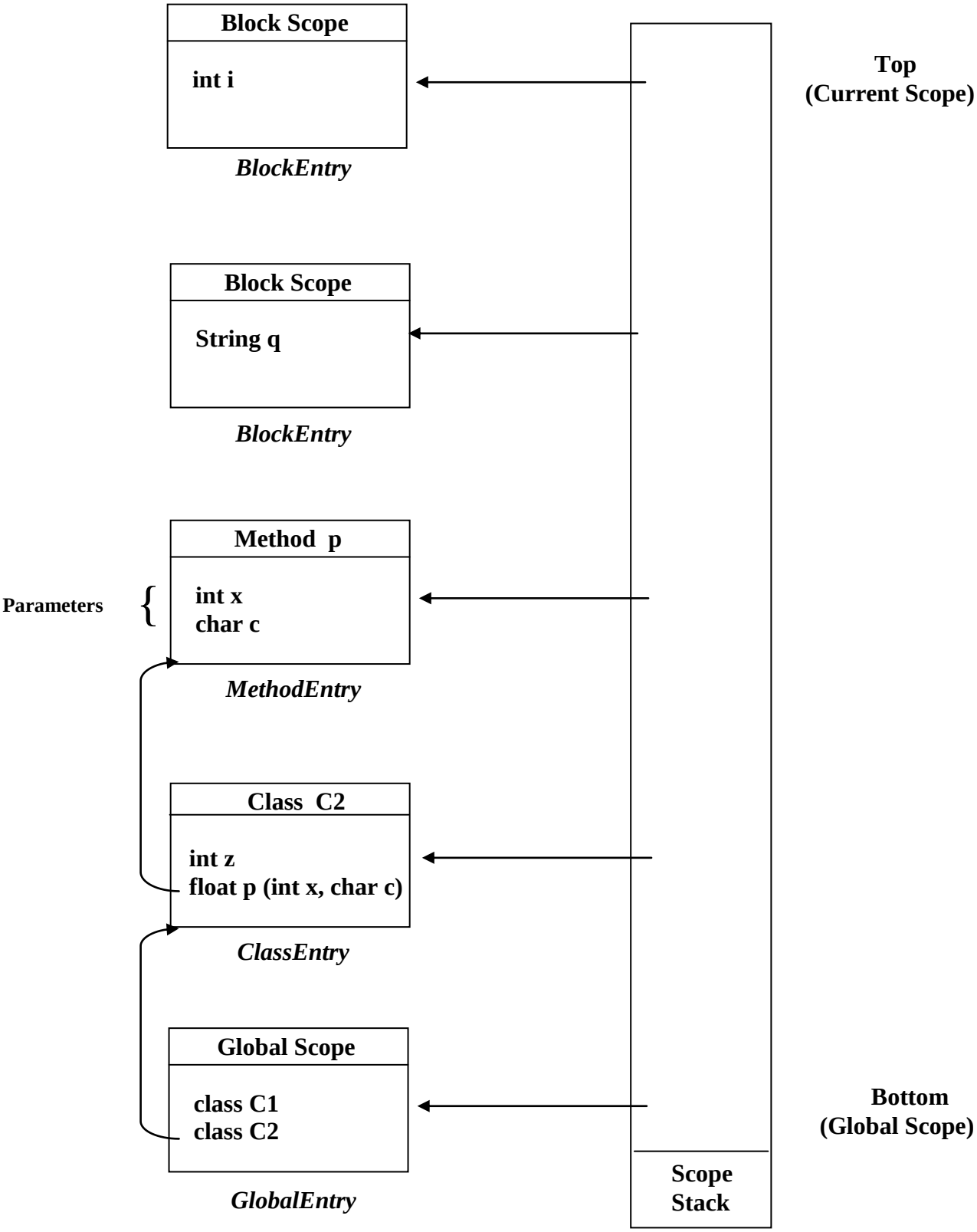


**Figure 4:** The scope entry objects in a *SymbolTable* object.

**Testing your program**

Since two of the subsequent labs will use your classes from this lab, the symbol table must be tested using the test data provided to make sure it works correctly. Also, the test data gives some examples of what an input file would look like if you want to create your own test cases. The symbol table commands used to test your program are as follows:

**class** *identifier*

Declares a class named *identifier*, i.e., it inserts a binding for that class into the current scope. An error message is printed if the command is unsuccessful (there is already a binding for that name).

**method** type *identifier* ( **formals** )

Declares a method named *identifier* with the given parameters, i.e., it inserts a binding for that method into the current scope. An error message is printed if the command is unsuccessful (there is already a binding for that name).

**variable** type *identifier*

Declares a variable named *identifier* with the given type and inserts a binding for that variable in the current scope. An error message is printed if the command is unsuccessful (there is already a binding for that name).

**lookup1** *identifier*

Looks for a binding for this *identifier* in the symbol table. If a binding is not found, an error message is printed.

**lookup2** *identifier* **.** *identifier*

Looks in the symbol table for a binding with the qualified name *identifier***.***identifier*. The first identifier must be a class and the second must be a binding (variable or method) declared within that class.

**new_scope** *identifier*

Looks up *identifier* in the symbol table. If an entry is found, then that entry becomes the current scope, i.e., that entry is pushed onto the scope stack. If neither a method nor a class with that name is found, an error message is printed.

**end_scope**

Removes the current scope entry from the scope stack, i.e., it invokes a symbol table method that pops the current scope entry off the scope stack (i.e., ***leaveScope***). However, your symbol table program must never leave the global scope, i.e., it must NOT allow the global scope to be popped off the scope stack (see the symbol table skeleton file for more details).

**enclosing_method**

Searches the symbol table for the enclosing method scope and returns this *MethodEntry*. If a *MethodEntry* is one of the entries in the scope stack, then this will be the enclosing method scope, i.e., it will be one of the *ScopeEntry* objects on the scope stack. For example, in **Figure 4**, this command would find and return the method entry for **p**. If there is no *MethodEntry* in the scope stack, an error message is printed.

**new_block**

Creates a new *BlockEntry* object and makes it the current scope (on top of the scope stack), i.e., it enters a new block scope.

**print_symtab**

Prints the symbol table, i.e., the contents of the global scope. Recall that the bottom entry of the symbol table must always be the *GlobalEntry* object.

The syntax for the non-terminals, type, formal, and formals, is as follows:

type → *int* | *char* | *String* | *boolean* | *float* | *void*

formals →
| formal_list
formal_list → formal
| formal_list **,** formal
formal → type *identifier*

**Error messages**

The program **Main.java** is a handcrafted interpreter for the above command language. It tests your program by calling the methods you defined, i.e., the methods that have to be defined in the entry classes and symbol table class.

Your program should NOT have to print any additional error messages. All error messages will be printed by this interpreter program. Your program only has to return false when an operation is unsuccessful and true when successful. In some cases, null is returned when an operation is unsuccessful (e.g., lookups). The skeleton files give more specific details about when to return true and when to return false or null.

**A Specific Test Case**

The symbol table shown in **Figure 4** can be built by the sequence of commands given below. The last four commands cause either a variable, method, and/or class to be looked up in the symbol table (e.g., a class is looked up when looking up qualified names like **C1.s**).

```
class C1
new_scope C1
variable int s
method int m (char t)
new_scope m
new_block
variable String s
end_scope
end_scope
method void n (int x, int y)
new_scope n
new_block
variable char m
end_scope
end_scope
end_scope
class C2
new_scope C2
variable int z
method float p (int x, char c)
new_scope p
new_block
variable String q
lookup1  x
new_block
variable int i
lookup1  z
lookup1  x
lookup2  C1.s
lookup1  x
lookup1  x
lookup1  i
end_scope
end_scope
end_scope
end_scope
```

You should spend some time studying **Figures 3** and **4** and the above sequence of commands. Make sure you understand the correspondence between the commands and the structure of the program fragment in **Figure 3**. In Lab 4 you will be writing a program that builds such symbol tables while the program is being traversed. Therefore, you are required to gain an understanding of how the symbol table for a program is built using the above command language. The mid-term will have at least one question that requires that you know how to build a symbol table for a given program using this command language.

Also, bindings must be recorded in ScopeEntry objects so these bindings can be retrieved quickly and efficiently because they will be looked up frequently during type checking. Therefore, some form of a hash table needs to be used, such as a *HashMap* in Java.

However, *ScopeEntry* objects must also provide the ability to retrieve and print the inner entry components in the order they occur in the program while, at the same time, maintaining the lookup speed obtained through the use of a *HashMap* lookup table. That is, you will need to store entries in a data structure that preserves the order in which

the bindings are inserted into the lookup table. Therefore, *LinkedHashMap* (a class in the Java *util* package) is the recommended data structure for the lookup tables (see the comments in the skeleton files for more details). Retrieval of entries in order will be accomplished through the "iterator pattern", i.e., methods, **reset(), hasMore(),** and **next()** in class *ScopeEntry*. See the skeleton files for more details.

**Getting Started**

Download the ZIP file **lab1.zip** from the course web page (www.cs.mum.edu/cs440). After unzipping **lab1.zip**, you will find several Java files and a **tests** subdirectory. The only classes that you will modify are the *SymbolTable* class and the classes shown in **Figure 1**. That is, *SymbolTable*, *Entry*, and all the subclasses of *Entry* should be included. These "skeleton" files define the interfaces for the classes that represent the various program entities. The method interfaces in these files must NOT be changed. You can add any instance variables or "helper" methods to any of the classes if you find that you need them, but the method headers and method interfaces provided must NOT be changed.

These skeleton files contain many comments and additional details about how to implement these classes, so read these comments before you start. Very little has to be done in the Entry classes; most of the changes have to be made in the *SymbolTable* class. Understanding how and why the symbol table works as well as the test cases is the main purpose of this programming assignment.

The other Java files are used by **Main.java** and must not be changed. **Main.java** is a main program for testing your symbol table classes. **Main.java** will not work unless all the methods in your symbol table and entry classes are defined properly and have the proper interface. Do NOT turn in **Main.java**, since your classes must work with the one provided. Any testing I do will be done with the one provided. Test data is also provided and can be found in the **tests** subdirectory, e.g., **in01**-**in09** of this subdirectory. To help with your testing, the output as it should occur is also given in the "out" files of the **tests** directory, e.g., **out01**-**out09**. Your results should closely match these files. To run the first test, type the following at the DOS command line (i.e., run Main with the test files as input):

**java Main tests\in01**

**What To Hand In**

Hand in a ZIP file containing your symbol table class definitions for (1) **Entry.java** (and all the **.java** files of its subclasses), (2) **SymbolTable.java**, and (3) the outputs your program produces using the test data provided. Finally, you must include a short README file giving the status of your program. The text of the README file must also be in the body of the email when you submit your lab assignment. That is, the text of the README file must be both in a file in the ZIP file submitted and in the body of the email (so I can quickly determine your status without opening the other file and so I don't have to create a file with your status information).

Do NOT include any **.class** files; your grade will automatically be reduced if any files are missing or if any extra files are included in your ZIP file (no exceptions). The grader should not have to download and unzip a lot of extra files like **Yylex.java** or **Entry.class**.