MAHARISHI UNIVERSITY OF MANAGEMENT

COMPUTER SCIENCE DEPARTMENT

COMP 440 – Compilers – DE

Programming Assignment 4

Phase 1 of the Type Checker
Building the Symbol Table

Before you begin lab assignment 4, read and make sure you understand the hand out that explains attribute grammars and syntax directed definitions and how to implement them using SableCC; the course web page has a link to this hand out. The lab 4 assignment specification assumes that you have read and understand that hand out.

Introduction

So far we have implemented the first two components of our simple compiler, i,e., the lexer and parser. Now that we have a parse tree from the parse phase, we can do various middle layer processing; this would typically include context sensitive analysis (such as type checking) and various kinds of optimization. Type checking makes sure that the program is valid, and optimization then looks for more compact and efficient representations.

The first phase of type checking will be to build a symbol table that can be used by the second phase of type checking. Two phases are necessary because our language allows an earlier part of the program to reference entities that are defined later, such as mutually recursive methods. Consider the following program.

```
class c1 {
    string s;
    bool isOdd(int n) {
        if (n == 0) {
            return false;
        } else {
            return c2.isEven(n - 1);
        }
    }
}
class c2 {
    bool isEven(int n) {
        if (n == 0) {
            return true;
        } else {
            return c1.isOdd(n - 1);
        }
    }
}
```

Figure 1: A CP program fragment.

Type checking a program such as the one in Figure 1 would not be possible unless the symbol table is built prior to the actual type checking of statements and expressions; this is because method **c1.isOdd** refers to **c2.isEven**, but **c2.isEven** is declared later in the program. Therefore, in this phase of the lab project you will have to define a class that builds a symbol table containing the bindings for every class and entity declared in the program. The *GlobalEntry* object represents the global scope and must contain bindings for the classes declared in the global scope; the *ClassEntry* object represents a class and must contain bindings for the fields and methods declared in that class; the *VariableEntry* object represents a variable and must contain its type and name; the *MethodEntry* object represents a method and must contain its name, return type, and the bindings for its parameters. The symbol table created in phase 1 can then be used by phase 2 of the type checker to determine the types of the identifiers used in expressions and statements in a CP program. To build the symbol table, you will need to use the classes you created in previous labs, i.e., the symbol table and "entry" classes from Lab 1 and the parser classes generated by SableCC in Labs 2 and 3.

Getting Ready To Start

The first step is to download and unzip the **lab4-ST-sample.zip** file from the **Sample Attribute Grammar** link on the course web site. In **lab4-ST-sample.zip** you will find three files in the ***symtab_testing*** directory: **ST_parser.scc**, **SymTabTester.java**, and **TesterDriver.java**. These files and classes are provided as examples of the kind of thing you will be doing in this lab assignment. **ST_parser.scc** is a SableCC grammar and parser specification for the command language used to test your symbol table and entry classes in Lab 1. **SymTabTester.java** is a class that walks parse trees generated by the grammar specified in **ST_parser.scc**; it is an interpreter for the command language of Lab1, i.e., it executes these commands and builds a symbol table (similar to the hand crafted interpreter included in Lab1, i.e., in **Main.java**). Note carefully the naming conventions used in naming the methods in **SymTabTester.java**; they correspond directly to the grammar specification given in **ST_parser.scc** (see also the SableCC documentation).

Make Sure You Are Ready To Start

To make sure you are ready to start Lab4, you can try out the interpreter for the command language of Lab1 by generating the parser and syntax tree classes using SableCC, i.e., generate the parser using the following command:

```
sablecc  ST_parser.scc
```

Next compile the generated classes using the following command.

```
javac TesterDriver.java
```

Now you can run the test cases from Lab1 as follows (the parser will only parse the command language syntax, i.e., the test files from Lab1):

```
java   TesterDriver   testfile_from_lab1
```

You will need to use your files from Lab1, but you also have to make sure you keep all of the SableCC packages generated from ST_parser.scc separate from your Lab4 SableCC packages.

Getting started with Lab4

First, download and unzip the **lab4.zip** file.

Second, copy your grammar specification from Lab3 into Lab4, i.e., copy **CP_parser.scc**.

Third, copy the classes you defined in Lab1, i.e., **SymbolTable.java, Entry.java,** and all the subclasses of **Entry** into your Lab4 directory.

Finally, copy the file **Type.java** that was provided in Lab1.

Lab 4 builds on the previous Labs 1, 2, and 3. The main task is to define a class that can be used to traverse the parse tree created by the parser generated from **CP_parser.scc**. Lab 4 must build the symbol table during this traversal. Therefore, you must define a class that can be used to traverse the parse tree and perform the actions specified in the table of Figure 2 below.

To do this all you need to know is which class to extend and which methods to override. The class to be extended is the **DepthFirstAdapter** class generated by SableCC and located in the analysis package (subdirectory); this class does a depth first traversal of the parse tree although it does not do anything during this traversal. To build the symbol table, you will have to override some of the methods of **DepthFirstAdapter**, that is, you will need to create a subclass that overrides these methods (**SymTabTester.java** gives many examples of the kinds of overrides necessary). The names of the methods that you will override depend on the names of the classes generated by SableCC. (See the attribute grammar handout for examples and an explanation of how to determine which methods to override.)

Syntax			Action
program	→	*class_decls*	
class_decls	→	*class_decls class_decl*	
	|	*class_decl*	
class_decl	→	*class_hdr* { member_decls }	Leave the current class scope (similar to the end_scope command in the Lab1 language)
class_hdr	→	class *id*	Create a new *ClassEntry* object for *id* and add a binding to the symbol table (like the class command). Then enter the scope of this new *ClassEntry* (like the new_scope command).
member_decls	→	*member_decls member*	
	|	*member*	
member	→	field	
	|	*method*	
	|	;	
field	→	*type id* ;	Create a new *VariableEntry* using id and type**.**t (use the **getOut** method to retrieve attribute t of type); then add a binding for this variable to the symbol table (this action is similar to the result of the variable command).
	|	*type id* = *expr* ;	The same as the previous action.
	|	type *id* [*int_lit*] ;	The same as the previous action except that this is a declaration of an array; thus you will need to call *makeArray* to convert the type to an array type using *int_lit* as the array size.
type	→	`float`	type**.**t = Type.floatVar (use method **setout()**)
	|	`int`	type**.**t = Type.intVar
	|	`char`	type**.**t = Type.charVar
	|	`bool`	type.t = Type.boolVar
	|	`string`	type**.**t = Type.stringVar
method_hdr	→	*type id*	Create a new *MethodEntry* using id and type**.**t; add a binding to the symbol table (similar to the method command). Then enter the scope of this method so the formal parameters can be added to this new *MethodEntry* object (done in action for formal parameter declarations below).
method_hdr	→	void *id*	Same as the previous action except the type is **Type.voidValue** from the *Type* class of Lab1.
formal	→	*type id*	The same action as for *field* above.
formal	→	*type id* [ ]	The same as *field* for arrays. The array size should be 0 in this case.
method	→	*method_hdr* (formals) block	Leave the scope of this method.
	|	*method_hdr* ( ) block	Leave the scope of this method.

Figure 2: An attribute grammar for building the symbol table for CP programs.

The above table defines an attribute grammar specifying the actions that must be taken during the parse tree traversal in order to build the symbol table.

Figure 2 above sometimes notes that actions are similar to those of commands from the language defined and used in Lab 1. You can look at the code provided in **SymTabTester.java** for examples of the method calls used by that interpreter for testing the SymbolTable. For example, the table specifies that after traversing the node representing the **class_hdr** rule, the action is similar (without printing "successful/unsuccessful command") to executing the two commands, **class *id*** and **new_scope *id***. That is, a new **ClassEntry** object for *id* must be created and a binding for it inserted into the current scope; then this **ClassEntry** object must become the current scope entry (i.e., the top of the scope stack in the symbol table). You will find it helpful for completing Lab4 if you understand the files in the ***symtab_tester*** subdirectory and how they are related to create an interpreter for the command language of Lab1.

You should also spend some time becoming familiar with the **Type** class provided; it will be used extensively in Lab 5. The method **makeArrayType** is used in the actions in Figure 2; this method converts a type into an array type, e.g., it converts the type **int** into an array of **int** variables; the argument is the size of the array if known or zero if the size is unknown (a formal parameter).

The actions specified in the above table are to be executed after traversing the node for that rule. (See the attribute grammar handout for an explanation of how to insert evaluation rules after traversing a syntax tree node.)

Defining the SymTabBuilder class

The instance variable **symtab** must be declared as shown in Figure 3. The field **symtab** is the symbol table being built during the tree traversal. A skeleton file is provided.

```
class SymTabBuilder extends DepthFirstAdapter {
    public SymbolTable symtab;
        // your evaluation rules (i.e., method overrides) go here.
}
```

Figure 3: A program fragment of the class to build the symbol table.

Testing

You must test your program using the test files provided. To run your program, copy **ParserDriver.java** from Lab 3 and modify it so it uses your new class **SymTabBuilder** to traverse the syntax tree (see also **TesterDriver.java** for an example). During the traversal it should build a symbol table containing all the class signatures for the whole program (it will do this if you have overridden and implemented the appropriate methods of **DepthFirstAdapter**). The attribute grammar handout gives an example showing how to walk a parse tree using the "visitor pattern". **TesterDriver.java** also provides an example of how to walk a parse tree.

After you finish traversing and building the symbol table, print it. If you have built the symbol table properly, the output will look almost the same as the input except for newlines, whitespace, and comments, etc. (ignored tokens will not be printed since they were skipped during the parsing process).

Potential Problems

The only problem that seems to come up very often occurs when the number of times that a new scope is entered (calls to method **enterScope**) is not exactly equal to the number of times the scope is exited (calls to **leaveScope**). If your program does not work, this is usually the reason. Notice carefully that there are two rules in the attribute grammar defined in Figure 2 where a new scope is entered (**class_hdr** and **method_hdr**) and there are two rules where a scope is exited (**class_decl** and **method_decl**). If variable declarations seem to disappear or appear in the wrong scope, then it is usually because the scope is not being managed properly.

The same kind of problem can occur if a scope is left too soon. Make sure that at the end of the tree traversal there is only one entry on the scope stack and that you have not popped a scope entry prematurely. This will prevent 90 percent of the problems that could come up when implementing lab 4.

Converting an integer literal to an integer value

In an array variable declaration, the attribute grammar says that we need to convert an integer literal (the size of the array) to an integer value. An integer literal should be a token in your parser specification. In SableCC, tokens have a method *getText()* that returns the lexeme for that token as a *String*. Then there is a static method *Integer.parseInt(String)* that converts a *String* to an *int* that can be used for the final conversion to an integer value.

Traversing the syntax tree

After you have copied your ParserDriver.java file into Lab4, you will need to modify it so the visitor class, **SymTabBuilder**, traverses the parse tree created in Lab3. This is done as follows:

```
Start tree = p.parse();                 // Parse the input and create a syntax tree.
SymTabBuilder symTabVisitor = new SymTabBuilder();  // create the visitor object
tree.apply(symTabVisitor);   // traverse the tree and build the symbol table.
```

The code in the **symTabVisitor** object builds the symbol table for the input program.

You need to handle arrays in *VariableEntry*

You need to modify the **toString** method of *VariableEntry* so it handles arrays. For example, you need to print the array dimensions when an identifier **A** denotes an array with size 10; it would be printed as follows: int A[10]

If the array size is zero, it would be printed as: int A[ ]

To determine whether or not a variable is an array type, call the method **isArrayKind()** on the variable's type. The array size is retrieved through the method **getArraySize()**.

What To Hand In

Hand in a ZIP file containing (1) your **CP_parser.scc** grammar specification, (2) your **SymTabBuilder** class, (3) your **Entry** class including all of its subclasses, e.g., **ScopeEntry**, etc., (4) your **SymbolTable** class, and (5) your test output files (it should be clear which output file corresponds with which input file). You must also include (6) a README file stating the status of your project and describing any problems (if any) your parser and tree walker have with the various test files. Missing or unnecessary files will reduce your grade.

Important Note

I have tried to give the basic structure and most of the high-level details of this lab so every student can do it without too much difficulty. However, it is just as important that everyone understand why building the symbol table is necessary for phase two of the type checker and in particular why and how the symbol table manages the scope of identifier names and how it handles nested scopes. While doing this lab, each student should make sure he/she understands why each action needs to be done for each production rule and particularly how and why the actions accomplish the goal of building the correct symbol table (since it does if followed exactly as given). The final exam will require that you take what you have learned and apply it in a completely different situation (perhaps a different and "new" language with nested scopes), but without most of the guidance being given in this lab. You probably will not do well on the final exam unless you understand how to manage scope using a symbol table and understand how to implement attribute grammars using the SableCC tool.