MAHARISHI UNIVERSITY OF MANAGEMENT

COMPUTER SCIENCE DEPARTMENT

COMP 440 – Compilers -- DE

Programming Assignment 5
Phase 2 of the Type Checker

Overview of the Project

So far we have implemented two of the front-end components of our simple compiler (i.e., the lexer and parser) and phase 1 of the type checker. Now that we have a symbol table from phase 1, we can do the type checking of expressions and statements. Parsing ensures that a program is syntactically correct, but the goal of type checking is to make sure the program does not have type errors.

In this assignment, you are to write methods and classes that implement phase 2 of the type checker for CP programs. Your main program (**ParserDriver**) must first call your CP parser. If the parse is successful, it must then call phase 1 and 2 of the type checker. Phase 1 creates the symbol table (Lab 4). Phase 2 uses this symbol table to type check expressions and statements that occur in method and variable declarations. The CP source program will be named on the compiler's command line and error messages will be written to standard output.

Because we want to learn the main ideas of how all this is done, with minimal time, you are given the basic structure of a type checking system for your compiler. This should significantly simplify what you need to do for this lab. In general, for each parse tree node (that represents a syntactic construct) a method has to be defined to do the type checking of that construct.

Getting Starting

The type checker must produce an error message for each type error in the program, and then return a count of the number of errors detected. The skeleton files in **lab5.zip** contain the basic structure of the type checker for CP. Look over the methods provided in the **TypeChecker.java** skeleton file to get the basic idea of how the type rules are enforced and how the **Type** class is used in type checking.

In general, for each production rule that needs to be type checked, a method must be defined in class **TypeChecker** to enforce the specific type rule for that construct. The type rules for CP constructs are given below. However, to make your task easier, a "helper" method has been provided in the **TypeChecker.java** skeleton file for many of those rules. The **TypeChecker** class is an extension of the **DepthFirstAdapter** class from the **analysis** package (similar to what you did in **SymTabBuilder** for Lab 4). The **TypeChecker** visitor class will decorate each expression in the syntax tree with its type. As in Lab 4, the names of the methods that have to be overridden depend on the names of the tokens and non-terminals given in your parser specification, i.e., in **CP_parser.scc**.

Types

In our type checker, a type will be represented by class **Type** (provided in Lab 1); this class has three data fields: *type*, *kind*, and *array size*. You should familiarize yourself with the file **Type.java**. Most of the methods of **Type** will be useful in the type checker, and you should be familiar with them before you start.

Possible values for the `type` field include **intType** (int), **boolType** (boolean), **charType** (char), **floatType** (float), **stringType** (string), **voidType** (void), and **unknownType** (Error). **voidType** is used to specify that a method has no return value. The **unknownType** is used to represent nodes that should have a type, but don't because of type errors.

Possible values for *kind* include **variableKind** (a local variable or field that may be assigned to), **valueKind** (an expression value, i.e., a value that may be read but not changed), and **arrayKind**.

Most combinations of `type` and `kind` represent something in CP. For example, the type of an **int** expression like "1+2" will have a `type` of **intType** and a `kind` of **valueKind**. However, an identifier that denotes a variable must have a type with a `kind` equal to **variableKind**; this indicates that it can be assigned to if it appears on the left-hand side of an assignment statement.

Type Checking Expressions

Many nodes represent constructs that are expected to have a type (such as expressions, literals, etc.); you must assign a type attribute to all such nodes of the parse tree. This type attribute will then be used in type checking larger expressions. For example, **Figure 1** shows a simple example of a type checking method for integer literals. This method simply sets the type attribute.

```
public void outAIntegerLiteral(AIntegerLiteral node) {
    setOut(node, Type.intValue);
}
```

Figure1: Type checking method for the following SableCC production rule:

literal = {integer} int_literal

What to to do First-(level 1a)

Step (1)

Start by writing the type checking method for the **class_hdr** production rule.

class_hdr --> class *id*

In Lab4 you created a new *ClassEntry* object for the class being declared and inserted a binding for that class in the global scope. Also this *ClassEntry* should contain bindings for every method and field declared in that class since they would have been inserted by **SymTabBuilder** (Lab4). However, in this assignment (Lab5) your program must look up the class name (*id*) in the symbol table (since it should already be in the global scope of the symbol table) and your program must enter the scope of that *ClassEntry* object; this is so everything declared in that class will be visible during the type checking of that class.

Step (2)

Next write the method that leaves the scope of that class, i.e., the method for the **class_decl** rule.

class_decl --> class_hdr { member_decls }

class_decl --> class_hdr { }

This method is the same as for lab4, i.e., the method must leave the scope of the class being declared.

Step (3)

Write the method to type check **field**.

field --> type *id* = expr ;

The initializing expression is optional in a field declaration. When there is an initializing expression, the type checker must ensure that expr can be assigned to a variable with the given type. That is, the type attribute of the initializing expression must be retrieved so it can be checked to make sure it is compatible with the type of the variable being declared. See the attribute grammar handout (Figure D) for an example of how to retrieve an attribute (using *getOut()*) after a syntax tree node has been traversed.

To type check a field node, you will need to define the method **isAssignable**; the interface for that method is provided in the skeleton file **TypeChecker.java**. This method was placed in class **TypeChecker** because it is useful for type checking other constructs (like assignment statements and method calls). Method **isAssignable** returns true or false depending on whether or not the **source** type is assignable to the **target** type (it does not emit error messages). The types of an assignment statement's left-hand (target) and right-hand (source) sides must be identical, except when the target has type **float** or **int**; this is because we allow coercion of **int** to **float**, and we also allow **char** to be coerced to **int** or **float** in an assignment statement. Furthermore, the left-hand side must always be a type with variable kind, whereas the right hand side may have either a variable or value kind. Entire arrays may only be assigned if both sides are arrays and both have the same component type. The table in **Figure 2** gives the rules for implementing method **isAssignable** (any combination not shown in that table is an error and false must be returned).

To summarize, the type checker must make sure the type of the source expression is compatible with the type of the variable being declared; this is to be done using the method **isAssignable** as specified in **Figure 2**.

The **TypeChecker** object has a field named **errorCount**; this field starts out as zero, but is incremented during type checking whenever a type error is detected (see the examples in **TypeChecker.java**). Thus, if the initializing expression is incompatible with the type of the variable, then "**errorCount**" must be incremented.

target	source	Return value (boolean)
any type with **valueKind**	all types and all kinds	false, since assignment is not allowed when the target is not a variable
arrayKind with type **t**	**arrayKind** with type **t**	true, since assignment is allowed as long as both arrays have identical component types
type **t** with **variableKind**	type **t** with **valueKind** or **variableKind**	true, since assignment of a value to a variable with the same type is always valid
float type with **variableKind**	**int** or **char** type with **valueKind** or **variableKind**	true, since we allow coercion when no information is lost (as in Java and C++)
int type with **variableKind**	**char** type with **valueKind** or **variableKind**	true, since **char** values can be coerced to **int** values without loss of information
any type with **variableKind**	**errorType**	true, since we assume the type is correct if the source expression has type errors; this avoids generating more error messages than necessary.

Figure2: How to implement the method:
public static boolean **isAssignable**(Type target, Type source) { … }

Step (4)

Next write the code to decorate the syntax tree of each CP literal expression with its type attribute (i.e., implement the necessary methods in the **TypeChecker** class). This means defining the tree decorating methods for the production rules given below. Figure 1 gives an example of how this is done (but obviously not all literals will have type **int** as in that example).

literal --> *int_lit*
literal --> *char_lit*
literal --> *str_lit*
literal --> *float_lit*
literal --> *true*
literal --> *false*

The type attribute of the type expressions below must also be set as in Lab4 (using *setOut()*).

type --> *int*
type --> *char*
type --> *string*
type --> *float*
type --> *bool*

Step (5)

It is just as important that all production rules for expressions like E2 --> E3 have their type attribute set properly; that is, the type attribute of E2 must be set to the type attribute of E3 (see the hand out on attribute grammars for examples of retrieving and setting attributes). Failing to set the attribute of the left-hand side of one of these rules will lead to a null pointer exception. There are at least 8 or 9 such rules that must be defined (assuming you properly specified the precedence and associativity of expressions in Lab 3). Again, the rules you are looking for are those expression productions with a single non-terminal on the right-hand side, i.e., base case (non-recursive) rules from lab 3.

Similarly, the rule "primary --> literal" must have the type of primary set to the same type as literal (which gets its type from Step (4) above). Make sure that for all such production rules, there is an overriding method in class **TypeChecker** that sets the attribute of the left-hand side with the attribute from the right-hand side, i.e., pass the attribute up the tree from the child (right side) to the parent node (left side). Failure to do so will result in a null pointer exception when you start testing.

Another source of null pointer exceptions is failure to set the attribute of the parenthesized expression, i.e.,

primary --> (expr)

In this rule, the type attribute of primary must be set to the type attribute from expr.

Notice that, so far, we are only type checking expressions; thus this step only applies to the production rules for expressions.

Step (6)
To type check a CP program, your driver program (**ParserDriver**) will have to first run the parser and phase 1 (same as in Lab4). After creating the symbol table, the parse tree must be type checked and decorated with its type attributes (this is similar to the tree walk done in Lab4 except a **TypeChecker** object is used as the visitor that traverses the tree instead of a **SymTabBuilder** object that was used in Lab4).

To do this, first copy the file **ParserDriver.java** from Lab4; modify it so it creates a **TypeChecker** object with the symbol table created in phase 1 as input to its constructor. Execute phase 2 of the type checker by walking the parse tree with this new instance of **TypeChecker** as the visitor.

In **TypeChecker.java**, **symtab** is the symbol table created during phase 1 (**SymTabBuilder**). (Obviously if the symbol table has not been properly built by your **SymTabBuilder** and in particular if the current scope entry is not the global scope, then your program will not work properly.)

The **ParserDriver** must also issue an error message if a method **Main.main** has not been declared in the global scope (i.e., look the method up in the symbol table); this is necessary because, similar to Java, execution of CP programs begin with this method.

The result of type checking will be a count of the number of type errors detected and this total must be printed. That's it! Now you should be ready to start testing.

Step (7)
Use the test files **Literals.cp** and **Literals_E.cp** (in subdirectory **level1a**) to test the checking of literals by your type checker. Your output should approximate the corresponding **.out** file. It is best not to continue until you have this part of the type checker working properly (this eliminates the most common sources of null pointer exceptions).

To make sure you have correctly implemented Step (5), i.e., handled parenthesized expressions, run the tests **Paren.cp** and **Paren_E.cp** in the **level1a** subdirectory.

Type Checking Identifiers - (level 1b)
In a CP program, an identifier may denote a class, field, method, parameter, or local variable. You will need to ensure that identifiers have been properly declared. To do this, you need to be able to uniquely detect each identifier, taking scope issues into account (the main scope issues will be handled later in level 3 of this assignment).

All identifiers, whether fields, methods, or local declarations, must be declared and must be visible within the context of their use. An error message must be printed if a use of an undeclared identifier is found; this can be done when type checking a "**field_access**" node; these nodes contain either one or two identifiers that can be looked up in the symbol table using methods already defined in Lab1. The symbol table is a data field in the **TypeChecker** "visitor" class for use when traversing the syntax tree; recall that the symbol table was created in Lab4 for use in Lab5. The two production rules for **field_access** are as follows:

field_access --> id
field_access --> id **.** id

The type checker must resolve the name in a **field_access** node by looking it up in the symbol table. For qualified names (as in the second rule), the first identifier must be a class declared in the program (as in Lab1). If a name is not found in the symbol table, the attribute of the **field_access** node must be set to **Type.errorType**. Setting the attribute to **Type.errorType** prevents propagating superfluous error messages. If the field name is found, then the attribute of that **field_access** node must be set to the type from the variable entry found in the symbol table.

Furthermore, if the entry found is not a **VariableEntry**, then the attribute must be set to **Type.errorType** and an error message must be issued (since the entry found during the lookup is not a variable).

To avoid null pointer exceptions, the type checking action must also be defined for the rule

primary --> field_access.

This rule is handled like those you handled in Step (5) of level 1a above (and may have already been done). The test cases for level 1b are found in the subdirectory **level1b**.

Type Checking Arithmetic Expressions - (level 2a)

You will have to write a "helper" method, **checkBinaryNumericOp**, for type checking arithmetic expressions (+, -, *, /, %); the skeleton of this method is provided in **TypeChecker.java**. These numeric operators can be applied to numeric operands, i.e., char, int, and float. The type of the whole arithmetic expression is determined after coercing one of the operands if necessary. For example, char can be coerced to int or float, and int can be coerced to float. Therefore, the type of the expression "2 + 4.5" would be float (after coercing '2' from int to float), but the type of expression "2 + 4" would be int (with no coercion).

In level 2a, you will also have to define the type checking methods for the unary arithmetic operators + and -. For these unary operators, and most other expressions and operators, all you have to do is look at the type rules that apply to the particular construct (given below) and find a method in **TypeChecker.java** that accomplishes the required type checking function (an example is given in the next section). To choose the appropriate helper method, always consider the required types of the operands and the type of the overall expression; from this (and the comments) it is easy to choose which method to use.

The next section explains with an example how to use the "helper" methods provided. You should become familiar with all the "helper" methods provided; this may also be helpful as preparation for the final exam (However, you only have to be familiar with what they do because I would provide you with the method interfaces, i.e., the method name and parameter types.)

```
public void outAAndLogicalAndExpression(AAndLogicalAndExpression node) {
        Node exp1 = node.getLogicalAndExpression();
        Node exp2 = node.getEqualityExpression();
        Type t1 = (Type)getOut(exp1);
        Type t2 = (Type)getOut(exp2);
        Type t = checkLogicalExpr(t1, t2, node.getAndOp());
        setOut(node, t);
}
protected Type checkLogicalExpr(Type t1, Type t2, Token op) {
        if (! t1.isBoolType() && ! t1.isErrorType()) {
           System.out.println("line " + op.getLine()
                           + ": invalid argument 1 to operator `"
                           + op.getText() + "' – expects a boolean argument");
           errorCount++;
        }
        if (! t2.isBoolType() && ! t2.isErrorType()) {
           System.out.println("line " + op.getLine()
                           + ": invalid argument 2 to operator `"
                           + op.getText() + "' – expects a boolean type");
           errorCount++;
        }
        return Type.boolValue;
}
```

Figure3: A program fragment showing the type checking method for the logical-and expression.

Type Checking Logical and Relational Expressions - (level 2b)

The skeleton file for the **TypeChecker** class, provided in **lab5.zip**, contains most of the "helper" type checking methods needed, i.e., methods like **checkLogicalExpr** shown in **Figure 3**; they can be used to check most of the different kinds of expressions. Therefore, checking expressions should be fairly easy. Your main task in level 2 (a and b) of the lab assignment will be to get the basic type checking of expressions working properly.

Figure 3 illustrates how to type check the following SableCC production rule:

logical_and_expression = {and} logical_and_expression and_op equality_expression

Note carefully that the naming conventions for the method **outAAndLogicalAndExpression** and method calls like **getAndOp()** and **getLogicalAndExpression()** are determined from the SableCC production rule, i.e., the terminals and non-terminals referenced by the rule. In this example, we are retrieving the type attribute of the two sub-expressions and then setting the synthesized attribute for the entire logical expression. That is, the type checking method makes sure that both sub-expressions are boolean expressions and then it sets the type attribute for the whole expression to boolean (the return type from **checkLogicalExpr**).

Use the test files in the **level2b** subdirectory to test the relational and logical expressions. Your output should approximate the corresponding **.out** file.

Preventing Cascading Error Messages – (levels 1 and 2)

To prevent one type error from propagating and causing superfluous error messages, your type checker should set the attribute of a node to **Type.errorType** whenever a type error is detected. Thus, to prevent cascading error messages, an error type should always be treated as if it were the correct type. For example, the following expression should cause only one error message to be issued and the whole expression should have type **int**:

```
(true + 3) + 4
```

Another example occurs when an identifier has not been declared. For example, the following expression should cause only one error message to be issued (the unresolved identifier message for identifier **a**) and the whole expression should have type **int**:

```
(a + 5)
```

Use the line numbers contained in the "Token" nodes to improve the specificity of your error messages; try to make them as informative as possible. Most of the error messages are already provided in the "helper" methods of the **TypeChecker** class. Notice the way that the "helper" methods use **Type.errorType** to avoid cascading error messages; you will be expected to do this in your implementation of methods **checkBinaryNumericOp** and **isAssignmable** as well as on the final exam.

Arrays - (level 3)

Consider the following declaration:

```
int  a[5];
```

The type of identifier **a** is array of int (i.e., int type and array kind), but the type of its components, e.g. **a[0], a[1]** etc., is an int variable, i.e. **Type.intVar** (int type and variable kind). Thus the type of an array component is the same as the array but it has variable kind rather than array kind. To convert from array kind to variable kind, call method **makeVariableType** of class **Type**, i.e., this converts an array kind to a variable kind without changing the `type` field. The type of an **array_ref** node must be determined in this way; however, if the kind of the variable entry is not an array kind, then an error message must be issued and the attribute is set to **Type.errorType**. To do this, you have to define the type checking rules for:

```
array_ref  -->  id  [  expr  ]
array_ref  -->  id . id  [  expr  ]
```

The variable names have to be resolved as in level 1b above.

Type Checking Method Calls - (level 3)

Type checking of method calls requires some care. In phase 1 (lab 4), you should have inserted bindings for each declared parameter in the **MethodEntry**. The "iterator" methods **first( ), next( ),** and **hasMore( )** (inherited from **ScopeEntry**) allow access to the parameters in the order they were declared. When a call is type checked you should build an ArrayList (or Vector) containing the types of the actual parameters of the call. To ensure that the correct number of arguments has been passed, make sure that the size of the Vector of actual parameter types is the same as the number of formal parameters. You must also compare the corresponding formal and actual parameter pairs to ensure that the type of each actual parameter is assignment compatible with its corresponding formal parameter, i.e., make sure the value of the actual parameter can be assigned to the formal parameter (use the helper **isAssignable** when checking each pair of parameters).

For example, if we had the declaration `int  p( int a, boolean b[ ] ){ ... }`

and the call `p(1, false);`

we would have a parameter list with types (Type.intVar, (Array of Type.boolVar)) for `p`'s declaration and the argument list with types (Type.intValue, Type.boolValue) for `p`'s call. Since a type with a value `kind` cannot be assigned to a type with an array `kind`, we can determine that the second parameter in `p`'s call is incorrect, i.e., false cannot be passed to a formal parameter that is an array.

Type Checking the Body of a MethodEntry - (level 4)

The scope rules of CP are similar to those of C++ and Java. A program consists of a sequence of named classes. All members within each class (fields and methods) are static (in the Java sense). Parameters of a method are considered local declarations visible within that method's body (and nowhere else). Local declarations within a block override any declaration in surrounding scopes, but one identifier name may be declared only once in any particular scope. You must print an error message if an identifier is re-declared within the same class, block, or sequence of method parameters. Duplicate declarations of fields and parameters will already have been reported during phase 1 of the type checker, i.e., during the building of the symbol table in Lab4. However, duplicate declarations within blocks must be detected in phase 2.

To implement the rest of the type checker you will need to use the scoping capabilities of your symbol table to enter and exit the scope of methods and blocks (similar to what you did in **level1a** for class declarations). Method scopes must be entered and exited in much the same way that class scopes were handled so that a method's parameters will be in scope (visible) when checking the body of that method. That is, your type checker must look up the method name and enter the scope of that method after traversing a **method_hdr** node, like you did for **class_hdr** in **level1a** above. Then similar to **class_decl**, your type checker must leave the method scope after traversing a **method** node.

Type Checking Blocks - (level 4)

When traversing the syntax tree node representing a block, an action must be performed prior to traversing that node and then a second action after. That is, you must enter the scope of a new **BlockEntry** before traversing that node and you must leave the block scope after. An easy way to enter a new block scope is to call method **enterNewBlock** of the symbol table. See the attribute grammar handout for an explanation of how to add an action prior to traversing a syntax tree node (i.e., you will have to override the "in" method of the block node). You need to enter a new block scope because the **BlockEntry** must contain all the bindings for the local variables declared in that block and they must not conflict with the bindings in outer scopes. Therefore, you must do the following when type checking a block node:

1. Enter a new block scope before traversing the syntax tree of the block node

2. Each time a **local_decl** node is encountered, your type checker must type check the node to make sure the initializing expression is compatible with type of the declared variable; it must also add a binding for each local declaration encountered in the block, i.e. add a binding to the symbol table (similar to what you did for **field** in Lab4)

3. Leave the block scope after traversing the block node (since these local declarations are no longer visible outside the block)

Once the type checker enters and exits class, method, and block scopes properly, the proper variable or method entry will be found when the type checker looks up identifiers in the symbol table (the implementation in level 1b should not have to change). Therefore, all uses of an identifier *x* will access the declaration corresponding to *x*, even though that declaration may be far removed from its use.

Type Checking Statements - (level 4)

Use the following rules to type check assignment, if-then-else, and while statements. Use the "helper" methods provided.

The Type Rules

The type rules of the CP language are enforced as follows:

- Unary operators + and - may only be applied to numeric operands (int, char, or float).
- Arithmetic operators (+, -, *, /, %) may be applied to numeric values (int, char, and float).
- The binary + operator may be applied to pairs of String values (as in Java).
- Logical operators (&& and ||) may be applied only to pairs of boolean values.

- Relational operators (==, <, >, !=, <=, >=) may only be applied to a pair of numeric values (int, char, or float), a pair of String values, or a pair of boolean values.
- Relational operators *can* be applied to boolean values; by definition (in CP), false is less than true.
- The type of a conditional expression (in an if or while construct) must be boolean.
- The types of an actual parameter and its corresponding formal parameter follow the same rules as assignment above (i.e., it must be valid to assign the actual parameter to corresponding the formal parameter).
- Array arguments may only be passed as array parameters that have the same component type.
- The type of a method call expression is the method's result type.
- A warning message should be issued if an identifier denoting a method with a result type other than **void** is called in a statement (the return value is being discarded).
- Return statements with an expression may only appear in a method with a result type other than **void**.
- The value returned by a return statement must have a type that is assignment compatible with the result type of the method within which it appears (i.e., it must be valid to assign the return value to a variable with the same type as the method's return type).
- Return statements without an expression may only appear in a method with a **void** result type.
- The size of an array (in a declaration) must be greater than zero.
- Only expressions of type int or char may be used to index arrays.
- The unary operator ! may only be applied to boolean values.

Type Checking Return Statements - (level 4)

See the type checking rules above. Use the symbol table method **enclosingMethod()** to find the return type of the enclosing method. The other helper method for checking return statements is **isAssignable** since the value of the return expression must be assignable to the return type of the method.

What to hand in

Hand in a ZIP file containing your entire project. You must hand in (1) your **CP_parser.scc** grammar specification, (2) your file containing the **TypeChecker** class, (3) your file containing the **SymTabBuilder** class, (4) your files containing all of the **Entry** classes (i.e., include **Entry** and all of **Entry**'s subclasses), (5) your file containing the **SymbolTable** class, (6) your file containing the **ParserDriver** class, and (7) your output files corresponding to the test files (it should be clear which output file corresponds with which input file). You must also include (8) a README file stating the status of your project and describing how your parser and tree walker behaves on the various test files (including which levels have been fully tested and which have been programmed but not fully tested). Do NOT include any **.class** files; only source files should be included. Missing or unnecessary files will reduce your grade. Finally, (9) you must include your file containing the **Type** class but ONLY IF you made any changes to it (usually not necessary).

Your README file must state clearly which levels have been completed and how far you got in testing your type checker using the test files provided (e.g., it is possible to finish and test part of level 2).

The quality of your documentation is also important. Make sure that you provide both external (in the README file) and internal (program comments) documentation. It should be easy for the grader to understand the organization and structure of all your programs.

Project Evaluation

The maximum grade on the lab portion of the course will be determined approximately as follows:

Max Lab Grade	Progress on the Lab Project
C+/B-	Completed lab 4 (about 100 lines of code) The symbol table builder class.
B/B+	Completed level 1a of lab 5 (about 100 lines of code-about 1/3 the same as lab4) Setting the type attribute of literal expressions and type checking variable declarations.
B+/A-	Completed level 1b of lab 5 (about 40 lines of code) Type checking identifier references.
A-	Completed level 2a of lab 5 (about 100 lines of code) Type checking unary and binary arithmetic operators (+, -, *, /, %).
A/A-	Completed level 2b of lab 5 (about 40-50 lines of code) Type checking logical and relational operators. This level of the project is the simplest part of the project (you just retrieve the attributes of the children nodes and use them to compute the attribute of the parent by calling one of the helper methods provided).
A	Completed level 3 of lab 5 (about 100+ lines of code) Type checking array references and method calls.
A+	Completed level 4 of lab 5 (about 100+ lines of code) Type checking method declarations, i.e. managing method and block scope so statements can be type checked (e.g., assignment, if-then-else, while, and return statements).

As preparation for the final exam, it is best to have completed levels (1a), (1b), and (2a) at a minimum; these are examples of some of the kinds of exam questions to expect. Level 3 is often helpful as well, that is, even if you do not have time to do level 3 or 4 of the project, make sure you understand how to type check array references, method calls, and assignment statements, and how to control the scope of identifiers (level 4) since this understanding will be helpful for the final.

Remember that your progress (or lack thereof) is only used to determine your maximum possible score on the lab part of the grade (the above are only tentative and approximate goals, i.e. they could change slightly). However, your overall lab grade is ultimately based on how well you can answer questions related to the lab project, not on the project you turn in. That is, your lab grade is the lower of the two scores. Therefore, to do well on the lab part of the course you must both do the labs and understand them. It is more important that you understand what needs to be done than that you eliminate all the bugs in your program and finish the project (I always favor the score on the exam rather than the level of progress on the lab turned in because it is a stronger indication of what the student can accomplish without help from others). Furthermore, it is useless to copy another person's lab because you will not be able to answer the kinds of questions that will be on the exams nor will you be able to finish the exam. This is because you will find yourself wasting time trying to figure out things that would already have been understood when doing the lab assignments. The final exam requires that you review everything you've done in the lab so you can quickly apply what you've learned.

The lab grade is worth 30-35% of the course grade.