```
package decaf;

import java.lang.reflect.Method;
import java.util.List;
import java.util.Stack;
import org.antlr.v4.runtime.Token;
import org.antlr.v4.runtime.tree.TerminalNode;

import decaf.DecafParser.Arg_typeContext;
import decaf.DecafParser.Meth_typeContext;
import decaf.DecafParser.TypeContext;

public class ScopeListener extends DecafParserBaseListener {
        private Stack<Scope> scopes;
        public String currentMeth = null;
        public int currentReturn = DecafParser.VOID;
        public boolean returnFound = false;
        public int forsNested = 0;

        public ScopeListener() {
                scopes = new Stack<Scope>();
                scopes.push(new Scope(null));
        }

        /*
         * Make sure there is main method before exiting the program
         * (non-Javadoc)
         * @see decaf.DecafParserBaseListener#exitProgram(decaf.DecafParser.
             ProgramContext)
         */
        public void exitProgram(DecafParser.ProgramContext ctx) {
                // Rule 3: No "main" method without arguments
                Scope scope = scopes.peek();
                ScopeElement found = scope.find("main");
                if (found == null || found.getTypes() != null)
                        System.err.println("Error: Program does not contain
                            method \"main\" with no arguments.");
        }

        public void enterBlock(DecafParser.BlockContext ctx) {
                scopes.push(new Scope(scopes.peek()));
        }

        public void exitBlock(DecafParser.BlockContext ctx) {
                scopes.pop();
        }

        /*
         * Arrays must be handled in a special way
         * (non-Javadoc)
         * @see decaf.DecafParserBaseListener#enterArray_name(decaf.DecafParser.
             Array_nameContext)
         */
        public void enterField_name(DecafParser.Field_nameContext ctx) {
                String name = ctx.ID().getText();
                Token token = ctx.getStart();
                int line = token.getLine();
        DecafParser.TypeContext tctx = ((DecafParser.Field_declContext)
            ctx.getParent()).type();
        int type = (tctx.INT() == null) ? DecafParser.BOOLEAN :
            DecafParser.INT;

        TerminalNode number = ctx.NUMBER();

        Scope scope = scopes.peek();
        ScopeElement found = scope.find(name);

        // Rule 1: Redeclaring variable in same scope
        if (found != null)
                System.err.println("Error on line "+line+": Variable \""
                    +name+"\" is already declared in this scope.");
        else if (number != null)
        {
                int n = Integer.parseInt(number.getText());
                scopes.peek().put(name, new ScopeElement(name, type,
                    line, n));
        }
        else
                scopes.peek().put(name, new ScopeElement(name, type,
                    line));

}

/*
 * Calling a method must make sure there are no type mismatches
 * (non-Javadoc)
 * @see decaf.DecafParserBaseListener#enterMethod_call(decaf.DecafParser
     .Method_callContext)
 */
public void enterMethod_name(DecafParser.Method_nameContext ctx) {
        String name = ctx.ID().getText();
        Token token = ctx.getStart();
        int line = token.getLine();

        Scope scope = scopes.peek();
        ScopeElement found = scope.find(name);

        // Rule 5: Check the method exists
        if (found == null)
        {
                System.err.println("Error on line "+line+": Method \""+
                    name+"\" has not been declared.");
                return;
        }

        // Rule 5: Check the number of arguments
        List<DecafParser.ExprContext> mtctx = ((DecafParser.
            Method_callContext)ctx.getParent()).expr();

        // Check if there are enough, if not return
        if (found.getTypes() != null)
        {
                if (found.getTypes().length != mtctx.size())
        {
                System.err.println("Error on line "+line+":
                    Mismatch in number of arguments for method \"
                    "+name+"\". Expected "+found.getTypes().
                    length+", received "+mtctx.size()+".");
                return;
        }
}
else if (mtctx.size() > 0)
{
        System.err.println("Error on line "+line+": Mismatch in
            number of arguments for method \""+name+"\". Expected
             "+found.getTypes().length+", received "+mtctx.size()
            +".");
        return;
}

// If there are enough, break the types into a list
int[] argTypes = new int[mtctx.size()];
for (int i = 0; i < mtctx.size(); i++)
{
        String[] parts = mtctx.get(i).getText().split("\\("); //
             Make sure method arguments aren't included
        ScopeElement argFound = scope.find(parts[0]);
        if (argFound != null)
        {
                argTypes[i] = argFound.getType();
        }
        else
        {
                if((mtctx.get(i).literal() != null))
                {
                        try
                        {
                                Integer.parseInt(mtctx.get(i).
                                    literal().NUMBER().getText())
                                    ;
                                argTypes[i] = DecafParser.INT;
                        } catch (Exception e)
                        {
                                argTypes[i] = DecafParser.
                                    BOOLEAN;
                        }
                }
                // If it's a mathematical operation, must be an
                    int value.
                // The values passed into the expression will be
                     checked during the
                // expression validation
                else if (mtctx.get(i).PLUS() != null || mtctx.
                    get(i).MINUS() != null ||
                                mtctx.get(i).MULT() != null ||
                                    mtctx.get(i).DIV() != null ||
                                mtctx.get(i).MOD() != null)
                {
                        argTypes[i] = DecafParser.INT;
                }
```

```
                        // Same if it's a comparative operation, must be
                             a boolean value.
                        // Again, values in expression will already be
                            checked at this point
                        else if (mtctx.get(i).NOT() != null || mtctx.get
                            (i).rel_op() != null ||
                                        mtctx.get(i).eq_op() != null ||
                                            mtctx.get(i).AND() != null ||
                                        mtctx.get(i).OR() != null)
                        {
                                argTypes[i] = DecafParser.BOOLEAN;
                        }
                        // Otherwise, it's in parentheses, so dive down
                            a level and try again
                        else
                        {
                                mtctx.set(i, mtctx.get(i).expr().get(0))
                                    ;
                                i--;
                        }

                }
        }

        // Now compare the arguments
        for (int i = 0; i < argTypes.length; i++)
        {
                if (argTypes[i] != found.getTypes()[i])
                        System.err.println("Error on line "+line+": Type
                             mismatch calling method \""+name+"\".");
        }

}

/*
 * For entering methods into the scope
 * (non-Javadoc)
 * @see decaf.DecafParserBaseListener#enterMeth_name(decaf.DecafParser.
     Meth_nameContext)
 */
public void enterMeth_name(DecafParser.Meth_nameContext ctx) {
        String name = ctx.ID().getText();
        Token token = ctx.getStart();
        int line = token.getLine();

        // Declaration of type can be bool, int or void
        DecafParser.TypeContext tctx;
        int methType = DecafParser.VOID;
        try
        {
                tctx = ((DecafParser.Method_declContext)ctx.getParent())
                    .meth_type().type();
                if (tctx.INT() != null)
                        methType = DecafParser.INT;
                else if (tctx.BOOLEAN() != null)
                        methType = DecafParser.BOOLEAN;
        } catch (Exception e) {}
        // Make a list of argument types
        List<Arg_typeContext> mtctx = ((DecafParser.Method_declContext)
            ctx.getParent()).arg_type();
        List<TerminalNode> atctx = ((DecafParser.Method_declContext)ctx.
            getParent()).ID();
        int[] argTypes = new int[mtctx.size()];
        for (int i = 0; i < mtctx.size(); i++)
        {
                int mtype = (mtctx.get(i).type().BOOLEAN() == null) ?
                    DecafParser.INT : DecafParser.BOOLEAN;
                argTypes[i] = mtype;
        }

        scopes.peek().put(name, new ScopeElement(name, methType, line,
            argTypes));
        // Add the arguments to the scope
        // IMPORTANT: the new scope isn't added until the block begins,
            so these
        // must be removed on exit
        for (int i = 0; i < atctx.size(); i++)
        {
                scopes.peek().put(atctx.get(i).getText(), new
                    ScopeElement(atctx.get(i).getText(), argTypes[i],
                    line));
        }

        this.currentMeth = name;
        this.currentReturn = methType;

}

/*
 * Return must match the type
 * (non-Javadoc)
 * @see decaf.DecafParserBaseListener#enterReturn(decaf.DecafParser.
    ReturnContext)
 */
public void enterReturn(DecafParser.ReturnContext ctx) {
        this.returnFound = true;
        Token token = ctx.getStart();
        int line = token.getLine();
        if (this.currentMeth == null)
        {
                System.err.println("Error on line "+line+": Return
                    statement found outside a method.");
                return;
        }
        if (this.currentReturn == DecafParser.VOID && ctx.expr() != null
            )
        {
                System.err.println("Error on line "+line+": Method
                    should not return a value.");
                return;
        }
        try
        {
```

```
if (ctx.expr().literal() != null)
{
        if (ctx.expr().literal().NUMBER() == null &&
            this.currentReturn == DecafParser.INT)
                System.err.println("Error on line "+line
                    +": Method expects an integer return
                    type.");
        else if (ctx.expr().literal().bool_literal() ==
            null && this.currentReturn == DecafParser.
            BOOLEAN)
                System.err.println("Error on line "+line
                    +": Method expects a boolean return
                    type.");
}
if ((ctx.expr().MINUS() != null || ctx.expr().PLUS() !=
    null || ctx.expr().MULT() != null
                || ctx.expr().DIV() != null || ctx.expr
                    ().MOD() != null) && this.
                    currentReturn != DecafParser.INT)
{
        System.err.println("Error on line "+line+":
            Invalid return value.");
        return;
}
if ((ctx.expr().rel_op() != null || ctx.expr().NOT() !=
    null || ctx.expr().eq_op() != null ||
                ctx.expr().AND() != null || ctx.expr().
                    OR() != null) && this.currentReturn
                    != DecafParser.BOOLEAN)
{
        System.err.println("Error on line "+line+":
            Invalid return value.");
        return;
}
if (ctx.expr().location() != null)
{
        Scope scope = scopes.peek();
        String varName = ctx.expr().location().ID().
            getText();
        ScopeElement found = scope.find(varName);
        if (found != null)
        {
                if (found.getType() != this.
                    currentReturn)
                {
                        System.err.println("Error on
                            line "+line+": Invalid return
                             value.");
                        return;
                }
        }
        else
        {
                System.err.println("Error on line "+line
                    +": Call to undeclared variable "+
                    varName+".");
                return;
```

```
                    }
                }
            } catch (Exception e) {

            }
    }

    /*
     * Method arguments do not fall into a new scope so must be removed on
         exit
     * (non-Javadoc)
     * @see decaf.DecafParserBaseListener#exitMethod_decl(decaf.DecafParser.
         Method_declContext)
     */
    public void exitMethod_decl(DecafParser.Method_declContext ctx) {
            List<TerminalNode> atctx = ctx.ID();
            for (int i = 0; i < atctx.size(); i++)
            {
                    scopes.peek().remove(atctx.get(i).getText());
            }
            if (this.currentReturn != DecafParser.VOID && this.returnFound
                == false)
            {
                    System.err.println("Error: Method "+currentMeth+" does
                        not contain return the expected type.");
            }
            this.currentMeth = null;
            this.currentReturn = DecafParser.VOID;
            this.returnFound = false;
    }


    /*
     * If statements must be boolean checks
     * (non-Javadoc)
     * @see decaf.DecafParserBaseListener#enterIf(decaf.DecafParser.
         IfContext)
     */
    public void enterIf(DecafParser.IfContext ctx) {
            Token token = ctx.getStart();
            int line = token.getLine();

            Scope scope = scopes.peek();

            // Rule 5: Get all the expressions
            DecafParser.ExprContext exprs = ctx.expr();

            // Dig down to the bottom
            boolean bottom = false;
            while (bottom == false)
            {
                    if (exprs.LBRACE() != null)
                            exprs = exprs.expr(0);
                    else
                            bottom = true;
            }
```

```
        if(exprs.literal() != null)
        {
                if (exprs.literal().NUMBER() != null)
                {
                        System.err.println("Error on line "+line+":
                            Condition is not boolean.");
                        return;
                }
        }
        // If it's a location, check the data type of the array
        else if (exprs.location() != null)
        {
                String loc = exprs.location().ID().getText();
                ScopeElement found = scope.find(loc);
                if (found == null)
                        return;
                if (found.getType() != DecafParser.BOOLEAN)
                {
                        System.err.println("Error on line "+line+":
                            Condition is not boolean.");
                        return;
                }
        }
        // If it's a method, check the return type
        else if (exprs.method_call() != null)
        {
                String method = null;
                if (ctx.expr().method_call().CALLOUT() == null)
                        method = exprs.method_call().method_name().ID().
                            getText();
                ScopeElement found = scope.find(method);
                if (found == null)
                        return;
                if (found.getType() != DecafParser.BOOLEAN)
                {
                        System.err.println("Error on line "+line+":
                            Condition is not boolean.");
                        return;
                }
        }
        // If it's a mathematical operation, must be an int value.
        // The values passed into the expression will be checked during
            the
        // expression validation
        else if (exprs.PLUS() != null || exprs.MINUS() != null ||
                        exprs.MULT() != null || exprs.DIV() != null ||
                        exprs.MOD() != null)
        {
                System.err.println("Error on line "+line+": Condition is
                     not boolean.");
                return;
        }
}

/*
 * For loops must add to the counter
 * (non-Javadoc)
 * @see decaf.DecafParserBaseListener#enterFor(decaf.DecafParser.
     ForContext)
 */
public void enterFor(DecafParser.ForContext ctx) {
        this.forsNested++; // For checking breaks and continues are
            within a loop
}

public void exitFor(DecafParser.ForContext ctx) {
        this.forsNested--; // Decrement the counter
}

/*
 * Breaks and continues must only lie where the "forsNested" counter > 0
 * (non-Javadoc)
 * @see decaf.DecafParserBaseListener#enterBreak(decaf.DecafParser.
     BreakContext)
 */
public void enterBreak(DecafParser.BreakContext ctx) {
        Token token = ctx.getStart();
        int line = token.getLine();
        if (this.forsNested <= 0)
        {
                System.err.println("Error on line "+line+": Break
                    statement not within for loop.");
                return;
        }
}

public void enterContinue(DecafParser.ContinueContext ctx) {
        Token token = ctx.getStart();
        int line = token.getLine();
        if (this.forsNested <= 0)
        {
                System.err.println("Error on line "+line+": Continue
                    statement not within for loop.");
                return;
        }
}

/*
 * Variable declarations must check it doesn't already exist
 * (non-Javadoc)
 * @see decaf.DecafParserBaseListener#enterVar_name(decaf.DecafParser.
     Var_nameContext)
 */
public void enterVar_name(DecafParser.Var_nameContext ctx) {
        String name = ctx.ID().getText();
        Token token = ctx.getStart();
        int line = token.getLine();
        DecafParser.TypeContext tctx;

        // Field declarations and variable declarations can trip this
            rule
        // so be sure to cast to the correct type!
        try
        {
                tctx = ((DecafParser.Var_declContext)ctx.getParent()).
                    type();
        }
        catch (Exception e)
        {
                tctx = ((DecafParser.Field_declContext)ctx.getParent()).
                    type();
        }
        int type = (tctx.INT() == null) ? DecafParser.BOOLEAN :
            DecafParser.INT;

        Scope scope = scopes.peek();
        ScopeElement found = scope.find(name);

        // Rule 1: Redeclaring variable in same scope
        if (found != null)
                System.err.println("Error on line "+line+": Variable \""
                    +name+"\" is already declared in this scope.");
        else
                scopes.peek().put(name, new ScopeElement(name, type,
                    line));
}

/* += and -= must be int on both sides
 * (non-Javadoc)
 * @see decaf.DecafParserBaseListener#enterMath_assign(decaf.DecafParser
     .Math_assignContext)
 */
public void enterMath_assign(DecafParser.Math_assignContext ctx)
{
        Token token = ctx.getStart();
        int line = token.getLine();
        DecafParser.TypeContext tctx;
        DecafParser.AssignContext pctx = ((DecafParser.AssignContext)ctx
            .getParent().getParent());
        String var = pctx.location().getText();
        Scope scope = scopes.peek();
        ScopeElement found = scope.find(var);
        if (pctx.expr().literal().NUMBER() == null || found.getType() !=
             DecafParser.INT)
        {
                System.err.println("Error on line "+line+": Invald input
                     for mathematical operation.");
                return;
        }
}

/*
 * Assigning a value must make sure that the variable exists
 * (non-Javadoc)
 * @see decaf.DecafParserBaseListener#enterAssign(decaf.DecafParser.
     AssignContext)
 */
public void enterAssign(DecafParser.AssignContext ctx) {
        DecafParser.LocationContext lctx = ctx.location();
        String varName = lctx.ID().getText();
        Token token = ctx.getStart();
```

```
Scope scope = scopes.peek();
ScopeElement found = scope.find(varName);

// Rule 2: Call to undeclared variable
if (found == null)
{
        System.err.println("Error on line "+token.getLine()+":
            Call to undeclared variable \""+varName+"\".");
        return;
}

// If assigning to an array, check the value is of valid type
try
{
        if (ctx.location().expr() != null)
        {
                String locName = ctx.location().expr().getText()
                    ;
                ScopeElement foundLoc = scope.find(locName);
                if (foundLoc != null && foundLoc.getType() !=
                    DecafParser.INT)
                {
                        System.err.println("Error on line "+
                            token.getLine()+": Invalid location
                            in array.");
                        return;
                }
                else if (ctx.location().expr().literal().NUMBER
                    () == null)
                {
                        System.err.println("Error on line "+
                            token.getLine()+": Invalid location
                            in array.");
                        return;
                }
        }
} catch (Exception e) {}

// Make sure RHS is not an array
try
{
        String locName = ctx.expr().getText();
        ScopeElement foundLoc = scope.find(locName);
        if (foundLoc.getSize() > -1)
        {
                System.err.println("Error on line "+token.
                    getLine()+": Cannot assign an array.");
                return;
        }
} catch (Exception e) {}

// Must be boolean if one of these operators are used
if ((ctx.expr().NOT() != null || ctx.expr().AND() != null || ctx
    .expr().OR() != null ||
                ctx.expr().eq_op() != null || ctx.expr().rel_op
                    () != null) && (found.getType() ==
                    DecafParser.INT))
                {
                        System.err.println("Error on line "+token.getLine()+":
                            Type mismatch assigning to variable \""+varName+"\"."
                            );
                        return;
                }
                // Must be integer if one of these operators are used
                else if ((ctx.expr().PLUS() != null || ctx.expr().MINUS() !=
                    null || ctx.expr().MULT() != null ||
                                ctx.expr().DIV() != null || ctx.expr().MOD() !=
                                    null) && (found.getType() == DecafParser.
                                    BOOLEAN))
                {
                        System.err.println("Error on line "+token.getLine()+":
                            Type mismatch assigning to variable \""+varName+"\"."
                            );
                        return;
                }
                // If it is a method call, check the method type
                else if (ctx.expr().method_call() != null)
                {
                        ScopeElement method = null;
                        if (ctx.expr().method_call().CALLOUT() == null)
                                method = scope.find(ctx.expr().method_call().
                                    method_name().getText());
                        if (method == null)
                                return;
                        if (method.getType() != found.getType())
                        {
                                System.err.println("Error on line "+token.
                                    getLine()+": Type mismatch assigning to
                                    variable \""+varName+"\".");
                                return;
                        }
                }
                // If it's a location, check the type
                else if (ctx.expr().location() != null)
                {
                        ScopeElement location = scope.find(ctx.expr().location()
                            .ID().getText());
                        if (location == null)
                                return;
                        if (location.getType() != found.getType())
                        {
                                System.err.println("Error on line "+token.
                                    getLine()+": Type mismatch assigning to
                                    variable \""+varName+"\".");
                                return;
                        }
                }

        }

}
```