**class MType:**

*# Type of function declaration*

*# partype: parameter type*

*# rettype: return type*

def \_\_init\_\_(self, partype, rettype):

self.partype = partype

self.rettype = rettype

**class ExpCheck:**

**@staticmethod**

**def isNotIntFloat(expType):**

return True if type(expType) not in [IntType, FloatType] else False

**@staticmethod**

**def isOpIntFloat(operator):**

return True if str(operator) in ['+', '-', '\*', '/', '%', '>', '<', '>=', '<='] else False

**@staticmethod**

**def mergeCoerceType(lType, rType):**

return (FloatType(), False) if FloatType in [type(x) for x in [lType, rType]] else (IntType(), False)

**class Symbol:**

*'''*

*name: string*

*mtype: MType | ArrayType | ArrayPointerType | PrimitiveType*

*kind: Function() | Parameter() | Variable()*

*isGlobal: boolean*

*'''*

*# set the default declare type is FuncDecl*

**def \_\_init\_\_(self, name, mtype, value=None, kind=Function(), isGlobal=False):**

self.name = name

self.mtype = mtype

self.value = value

self.kind = kind

self.isGlobal = isGlobal

**def getKind(self):**

return self.kind if type(self.mtype) is MType else Identifier()

**def toTuple(self):**

return (str(self.name), type(self.getKind()))

**def toParam(self):**

self.kind = Parameter()

return self

**def toVar(self):**

self.kind = Variable()

return self

**def toGlobal(self):**

self.isGlobal = True

return self

**@staticmethod**

**def getNameString(symbol):**

return str(symbol.name)

**@staticmethod**

**def toVarDecl(decl):**

return Symbol(decl.variable, decl.varType, kind=Variable())

**@staticmethod**

**def toFuncDecl(decl):**

paramType = [x.varType for x in decl.param]

return Symbol(decl.name.name, MType(paramType, decl.returnType), kind=Function())

**@staticmethod**

**def toDecl(decl):**

return Symbol.toVarDecl(decl) if type(decl) is VarDecl else Symbol.toFuncDecl(decl)

**class ScopeCheck:**

**@staticmethod**

**def isExistIn(listSymbols, symbol):**

return len([x for x in listSymbols if str(x.name) == str(symbol.name)]) > 0

**@staticmethod**

**def merge(oldScope, newScope):**

return reduce(lambda lst, sym: lst if ScopeCheck.isExistIn(lst, sym) else lst+[sym], oldScope, newScope)

**class Checker:**

utils = Utils()

**@staticmethod**

**def checkRedeclared(currentScope, listNewSymbols):**

newScope = currentScope.copy()

for x in listNewSymbols:

f = Checker.utils.lookup(Symbol.getNameString(x), newScope, Symbol.getNameString)

if f is not None:

raise Redeclared(x.kind, x.name)

newScope.append(x)

return newScope

**@staticmethod**

**def checkUndeclared(visibleScope, name, kind, notGlobal=False):**

scope = visibleScope if not notGlobal else [x for x in visibleScope if not x.isGlobal]

res = Checker.utils.lookup(name, scope, lambda x: x.name)

if res is None:

raise Undeclared(kind, name)

return res

**@staticmethod**

**def matchType(requiredType, paramType):**

if type(requiredType) == ArrayPointerType and type(paramType) in [ArrayType, ArrayPointerType]:

return Checker.matchType(requiredType.eleType, paramType.eleType)

if type(requiredType) == type(paramType):

return True

if type(requiredType) is FloatType and type(paramType) is IntType:

return True

return False

**@staticmethod**

**def checkParamType(required, params):**

if len(required) != len(params):

return False

return all([Checker.matchType(a, b) for a, b in zip(required, params)])

**@staticmethod**

**def handleReturnStmts(stmts):**

*# stmts: [(stmt, type)]*

for i in range(0, len(stmts)-1):

if Checker.isStopTypeStatment(stmts[i][1]):

raise UnreachableStatement(stmts[i+1][0])

return None if stmts == [] else stmts[-1][1]

**@staticmethod**

**def checkReturn(stmts):**

for i in stmts:

if type(i[0]) is Return:

return True

return None

**@staticmethod**

**def isReturnTypeFunction(retType):**

return type(retType) in [IntType, FloatType, BoolType, StringType, ArrayPointerType, ArrayType]

**@staticmethod**

**def isReturnType(retType):**

return Checker.isReturnTypeFunction(retType)

**@staticmethod**

**def isStopTypeStatment(retType):**

return Checker.isReturnType(retType) or type(retType) in [Break, Continue]

**class Graph:**

link = {}

visited = {}

invoked = {}

**@staticmethod**

**def initialize():**

Graph.link.clear()

Graph.visited.clear()

Graph.invoked.clear()

**@staticmethod**

**def addToGraph(u, v=None):**

u = str(u)

if type(Graph.link.get(u)) != list:

Graph.link[u] = []

Graph.visited[u] = False

Graph.invoked[u] = False

if v is None:

return

v = str(v)

if v != u and v not in Graph.link[u]:

Graph.link[u].append(v)

Graph.invoked[v] = True

**@staticmethod**

**def depthFirstVisit(u):**

u = str(u)

Graph.visited[u] = True

[Graph.depthFirstVisit(v) for v in Graph.link[u] if not Graph.visited[v]]

**@staticmethod**

**def getUnreachableNode():**

for u in Graph.link:

if not Graph.visited[u] and not Graph.invoked[u]:

return u

for u in Graph.link:

if not Graph.visited[u]:

return u

return None

**@staticmethod**

**def setVisited(listNodes):**

for u in listNodes:

Graph.visited[str(u)] = True

**class StaticChecker(BaseVisitor, Utils):**

global\_envi = [

Symbol("getInt",MType([],IntType())),

Symbol("putIntLn",MType([IntType()],VoidType())),

Symbol("putInt", MType([IntType()], VoidType())),

Symbol("getFloat", MType([], FloatType())),

Symbol("putFloat", MType([FloatType()], VoidType())),

Symbol("putFloatLn", MType([FloatType()], VoidType())),

Symbol("putBool", MType([BoolType()], VoidType())),

Symbol("putBoolLn", MType([BoolType()], VoidType())),

Symbol("putString", MType([StringType()], VoidType())),

Symbol("putStringLn", MType([StringType()], VoidType())),

Symbol("putLn", MType([], VoidType()))]

**def \_\_init\_\_(self,ast):**

self.ast = ast

**def check(self):**

Graph.initialize()

return self.visit(self.ast, StaticChecker.global\_envi)

**def checkMain(self, symbols):**

for i in symbols:

if i.name == 'main' and type(i.kind) == Function:

return i

return None

**def visitProgram(self, ast: Program, globalEnv):**

symbols = [Symbol.toDecl(x).toGlobal() for x in ast.decl]

scope = Checker.checkRedeclared(globalEnv, symbols)

entryPoint = self.checkMain(symbols)

if entryPoint is None:

raise NoEntryPoint()

listFuncDecl = globalEnv + [entryPoint] + [Symbol.toDecl(x) for x in ast.decl if type(x) is FuncDecl]

for x in listFuncDecl:

Graph.addToGraph(x.name)

Graph.setVisited([u.name for u in globalEnv])

[self.visit(x, scope) for x in ast.decl]

Graph.depthFirstVisit("main")

u = Graph.getUnreachableNode()

if u is not None:

symbol = self.lookup(u, listFuncDecl, Symbol.getNameString)

raise UnreachableFunction(symbol.name)

return None

**def visitFuncDecl(self, ast: FuncDecl, scope):**

list\_params = [self.visit(x,scope).toParam() for x in ast.param]

list\_vars = [self.visit(x, scope).toVar() for x in ast.body.member if type(x) is VarDecl]

list\_new\_symbols = list\_params + list\_vars

localScope = Checker.checkRedeclared([], list\_new\_symbols)

newScope = ScopeCheck.merge(scope, localScope)

stmts = [self.visit(x, (newScope, ast.returnType, False, ast.name.name)) for x in ast.body.member if not(type(x) is VarDecl)]

retType = Checker.handleReturnStmts(stmts)

if Checker.isReturnTypeFunction(ast.returnType) and not Checker.isReturnTypeFunction(retType):

raise FunctionNotReturn(ast.name.name)

**def visitVarDecl(self, ast, scope):**

return Symbol.toVarDecl(ast)

**def visitBlock(self, ast: Block, params):**

scope, retType, inLoop, funcName = params

list\_vars = [self.visit(x, scope).toVar() for x in ast.member if type(x) is VarDecl]

localScope = Checker.checkRedeclared([], list\_vars)

newScope = ScopeCheck.merge(scope, localScope)

stmts = [self.visit(x, (newScope, retType, inLoop, funcName)) for x in ast.member if not(type(x) is VarDecl)]

ret = Checker.handleReturnStmts(stmts)

if ret is None:

return (ast, None)

else:

return (ast, ret)

**def visitIf(self, ast: If, params):**

*# scope: current environment*

*# retType: function return type*

*# inLoop: is this stmt in a loop or not (for break / continue)*

*# funcName: function name*

scope, retType, inLoop, funcName = params

condType = self.visit(ast.expr, (scope, funcName))[0]

if type(condType) is not BoolType:

raise TypeMismatchInStatement(ast)

stmt1 = [self.visit(ast.thenStmt, params)]

stmt2 = None

retType2 = None

return2 = None

if not(ast.elseStmt is None):

stmt2 = [self.visit(ast.elseStmt, params)]

retType2 = Checker.handleReturnStmts(stmt2)

return2 = Checker.checkReturn(stmt2)

retType1 = Checker.handleReturnStmts(stmt1)

return1 = Checker.checkReturn(stmt1)

if (retType1 is None or retType2 is None) and (return1 is None or return2 is None):

return (ast, None)

if all([x not in [type(retType1), type(retType2)] for x in [Break, Continue]]):

return (ast, retType)

return (ast, Break())

**def visitFor(self, ast: For, params):**

scope, retType, inLoop, funcName = params

*# get expressions type*

exp1Type = self.visit(ast.expr1, (scope, funcName))[0]

exp2Type = self.visit(ast.expr2, (scope, funcName))[0]

exp3Type = self.visit(ast.expr3, (scope, funcName))[0]

*# check expression type*

if (False in [type(x) is IntType for x in [exp1Type, exp3Type]]) or (type(exp2Type) is not BoolType):

raise TypeMismatchInStatement(ast)

*# get stmt list*

stmt = [self.visit(ast.loop, (scope, retType, True, funcName))]

*# check for unreachable / get return stmt (if any)*

retType = Checker.handleReturnStmts(stmt)

return (ast, None)

**def visitDowhile(self, ast: Dowhile, params):**

scope, retType, inLoop, funcName = params

*# get condition expression*

condType = self.visit(ast.exp, (scope, funcName))[0]

*# check condition expression type*

if type(condType) is not BoolType:

raise TypeMismatchInStatement(ast)

*# visit statement list*

stmt = [self.visit(x, (scope, retType, True, funcName)) for x in ast.sl]

*# check for unreachable / get return stmt*

retType = Checker.handleReturnStmts(stmt)

if retType is None:

return (ast, None)

else:

return (ast, retType)

**def visitContinue(self, ast, params):**

scope, retType, inLoop, funcName = params

if not inLoop:

raise ContinueNotInLoop()

return (ast, Continue())

**def visitBreak(self, ast, params):**

scope, retType, inLoop, funcName = params

if not inLoop:

raise BreakNotInLoop()

return (ast, Break())

**def visitReturn(self, ast: Return, params):**

scope, retType, inLoop, funcName = params

if type(retType) is VoidType and ast.expr:

raise TypeMismatchInStatement(ast)

ret = self.visit(ast.expr, (scope, funcName))[0] if ast.expr else VoidType()

if not Checker.matchType(retType, ret):

raise TypeMismatchInStatement(ast)

return (ast, ret)

*# visit func call*

**def visitCallExpr(self, ast: CallExpr, params):**

*# handle statement*

if len(params) == 4:

scope, retType, inLoop, funcName = params

self.handleCall(ast, scope, funcName, Function())

return (ast, None)

*# handle expression*

else:

scope, funcName = params

symbol = self.handleCall(ast, scope, funcName, Function())

return (symbol.mtype.rettype, False)

**def visitBinaryOp(self, ast: BinaryOp, params):**

*# handle expression*

*# return result type of the binary op*

if len(params) == 2:

scope, funcName = params

op = str(ast.op)

lType = self.visit(ast.left, (scope, funcName))[0]

rType = self.visit(ast.right, (scope, funcName))[0]

*# ['+', '-', '\*', '/', '%', '>', '<', '>=', '<=']*

if ExpCheck.isOpIntFloat(op):

if ExpCheck.isNotIntFloat(lType) or ExpCheck.isNotIntFloat(rType):

raise TypeMismatchInExpression(ast)

elif op in ['+', '-', '\*', '/']:

if type(lType) == IntType and type(rType) == IntType:

return (IntType(), False)

else:

return ExpCheck.mergeCoerceType(lType, rType)

elif op == '%':

if FloatType in [type(lType), type(rType)]:

raise TypeMismatchInExpression(ast)

else:

return (IntType(), False)

else:

return (BoolType(), False)

*# ['&&', '||']*

elif op in ['&&', '||']:

if not(type(lType) == BoolType and type(rType) == BoolType):

raise TypeMismatchInExpression(ast)

else:

return (BoolType(), False)

*# ['==', '!=']*

elif op in ['==', '!=']:

if (type(lType) == type(rType)) and (type(lType) not in [ArrayType, ArrayPointerType, VoidType, FloatType]):

return (BoolType(), False)

else:

raise TypeMismatchInExpression(ast)

*# ['=']*

elif op == '=':

***### handle not left value***

isLeft = self.visit(ast.left, (scope, funcName))[1]

if isLeft == False:

raise NotLeftValue(ast.left)

***###***

if type(lType) == IntType and type(rType) == IntType:

return (IntType(), False)

elif type(lType) == IntType and type(rType) == FloatType:

raise TypeMismatchInExpression(ast)

elif type(lType) == FloatType and type(rType) in [IntType, FloatType]:

return (FloatType(), False)

elif type(lType) == BoolType and type(rType) == BoolType:

return (BoolType(), False)

elif type(lType) == StringType and type(rType) == StringType:

return (StringType(), False)

else:

raise TypeMismatchInExpression(ast)

*# handle stmt*

else:

scope, retType, inLoop, funcName = params

op = str(ast.op)

lType = self.visit(ast.left, (scope, funcName))[0]

rType = self.visit(ast.right, (scope, funcName))[0]

*# ['+', '-', '\*', '/', '%', '>', '<', '>=', '<=']*

if ExpCheck.isOpIntFloat(op):

if ExpCheck.isNotIntFloat(lType) or ExpCheck.isNotIntFloat(rType):

raise TypeMismatchInExpression(ast)

elif op in ['+', '-', '\*', '/']:

if type(lType) == IntType and type(rType) == IntType:

return (ast, None)

else:

return ExpCheck.mergeCoerceType(lType, rType)

elif op == '%':

if FloatType in [type(lType), type(rType)]:

raise TypeMismatchInExpression(ast)

else:

return (ast, None)

else:

return (ast, None)

*# ['&&', '||']*

elif op in ['&&', '||']:

if not(type(lType) == BoolType and type(rType) == BoolType):

raise TypeMismatchInExpression(ast)

else:

return (ast, None)

*# ['==', '!=']*

elif op in ['==', '!=']:

if (type(lType) == type(rType)) and (type(lType) not in [ArrayType, ArrayPointerType, VoidType, FloatType]):

return (ast, None)

else:

raise TypeMismatchInExpression(ast)

*# ['=']*

elif op == '=':

***### handle not left value***

isLeft = self.visit(ast.left, (scope, funcName))[1]

if isLeft == False:

raise NotLeftValue(ast.left)

***###***

if type(lType) == IntType and type(rType) == IntType:

return (ast, None)

elif type(lType) == IntType and type(rType) == FloatType:

raise TypeMismatchInExpression(ast)

elif type(lType) == FloatType and type(rType) in [IntType, FloatType]:

return (ast, None)

elif type(lType) == BoolType and type(rType) == BoolType:

return (ast, None)

elif type(lType) == StringType and type(rType) == StringType:

return (ast, None)

else:

raise TypeMismatchInExpression(ast)

**def visitUnaryOp(self, ast: UnaryOp, params):**

*# handle expression*

if len(params) == 2:

scope, funcName = params

op = str(ast.op)

expType = self.visit(ast.body, (scope, funcName))[0]

if (op == '-' and ExpCheck.isNotIntFloat(expType)) or (op == '!' and type(expType) is not BoolType):

raise TypeMismatchInExpression(ast)

return (expType, False)

*# handle expression statement*

else:

scope, retType, inLoop, funcName = params

op = str(ast.op)

expType = self.visit(ast.body, (scope, funcName))[0]

if (op == '-' and ExpCheck.isNotIntFloat(expType)) or (op == '!' and type(expType) is not BoolType):

raise TypeMismatchInExpression(ast)

return (ast, None)

**def visitId(self, ast: Id, params):**

if len(params) == 2:

scope, funcName = params

symbol = Checker.checkUndeclared(scope, ast.name, Identifier())

return (symbol.mtype, True)

else:

scope, retType, inLoop, funcName = params

symbol = Checker.checkUndeclared(scope, ast.name, Identifier())

return (ast, None)

**def visitArrayCell(self, ast: ArrayCell, params):**

if len(params) == 2:

scope, funcName = params

arrType = self.visit(ast.arr, params)[0]

idxType = self.visit(ast.idx, params)[0]

if not(type(idxType) is IntType) or not(type(arrType) in [ArrayType, ArrayPointerType]):

raise TypeMismatchInExpression(ast)

return (arrType.eleType, True)

else:

scope, retType, inLoop, funcName = params

arrType = self.visit(ast.arr, (scope, funcName))[0]

idxType = self.visit(ast.idx, (scope, funcName))[0]

if not(type(idxType) is IntType) or not(type(arrType) in [ArrayType, ArrayPointerType]):

raise TypeMismatchInExpression(ast)

return (ast, None)

*# handle a function call*

**def handleCall(self, ast, scope, funcName, kind):**

symbol = Checker.checkUndeclared(scope, ast.method.name, Function())

paramType = [self.visit(x, (scope, funcName))[0] for x in ast.param]

if not Checker.checkParamType(symbol.mtype.partype, paramType):

raise TypeMismatchInExpression(ast)

Graph.addToGraph(funcName, ast.method.name)

return symbol

**def visitIntLiteral(self, ast, params):**

return (IntType(), False)

**def visitFloatLiteral(self, ast, params):**

return (FloatType(), False)

**def visitBooleanLiteral(self, ast, params):**

return (BoolType(), False)

**def visitStringLiteral(self, ast, params):**

return (StringType(), False)