1. [CC03] Type Programming Code 1

**[CC03] Type Programming Code 1**

|  |  |
| --- | --- |
| **Started on** | Wednesday, 3 April 2024, 11:24 AM |
| **State** | Finished |
| **Completed on** | Wednesday, 3 April 2024, 11:43 AM |
| **Time taken** | 19 mins 37 secs |
| **Marks** | 3.00/3.00 |
| **Grade** | **10.00** out of 10.00 (**100**%) |

Top of Form

### **Question**1

Correct

Mark 1.00 out of 1.00

Flag question

#### **Question text**

Given the AST declarations as follows:

class Exp(ABC): #abstract class

class BinOp(Exp): #op:str,e1:Exp,e2:Exp #op is +,-,\*,/,&&,||, >, <, ==, or  !=

class UnOp(Exp): #op:str,e:Exp #op is -, !

class IntLit(Exp): #val:int

class FloatLit(Exp): #val:float

class BoolLit(Exp): #val:bool

and the Visitor class is declared as follows:

class StaticCheck(Visitor):

    def visitBinOp(self,ctx:BinOp,o): pass

    def visitUnOp(self,ctx:UnOp,o):pass

    def visitIntLit(self,ctx:IntLit,o): pass

    def visitFloatLit(self,ctx,o): pass

    def visitBoolLit(self,ctx,o): pass

Rewrite the body of the methods in class StaticCheck to check the following type constraints:

* + , - and \* accept their operands in int or float type and return float type if at least one of their operands is in float type, otherwise, return int type
* / accepts their operands in int or float type and returns float type
* !, && and || accept their operands in bool type and return bool type
* >, <, == and != accept their operands in any type but must in the same type and return bool type

If the expression does not conform the type constraints, the StaticCheck will raise exception TypeMismatchInExpression with the innermost sub-expression that contains type mismatch.

Your code starts at line 55

**For example:**

| **Test** | **Result** |
| --- | --- |
| BinOp("+",IntLit(3),BoolLit(True)) | Type Mismatch In Expression: BinOp("+",IntLit(3),BoolLit(True)) |

Answer:(penalty regime: 0 %)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

class StaticCheck(Visitor):

def visitBinOp(self,ctx:BinOp,o):

type1 = self.visit(ctx.e1,o)

type2 = self.visit(ctx.e2,o)

if ctx.op in ['+','-','\*']:

if type1 == 3 or type2 == 3:

raise TypeMismatchInExpression(ctx)

if type1 == 2 or type2 == 2:

return 2

else:

return 1

if ctx.op == '/':

if type1 == 3 or type2 == 3:

raise TypeMismatchInExpression(ctx)

return 2

if ctx.op == '&&' or ctx.op == '||':

if type1 != 3 or type2 != 3:

raise TypeMismatchInExpression(ctx)

return 3

if ctx.op in ['>','<','==','!=']:

if type1 != type2:

raise TypeMismatchInExpression(ctx)

return 3

def visitUnOp(self,ctx:UnOp,o):

type0 = self.visit(ctx.e,o)

if ctx.op == '-':

if type0 == 3:

raise TypeMismatchInExpression(ctx)

return type0

if ctx.op == '!':

if type0 != 3:

raise TypeMismatchInExpression(ctx)

return type0

def visitIntLit(self,ctx:IntLit,o):

return 1

def visitFloatLit(self,ctx,o):

return 2

def visitBoolLit(self,ctx,o):

return 3

#### **Feedback**

|  | **Test** | **Expected** | **Got** |  |
| --- | --- | --- | --- | --- |
|  | BinOp("+",IntLit(3),BoolLit(True)) | Type Mismatch In Expression: BinOp("+",IntLit(3),BoolLit(True)) | Type Mismatch In Expression: BinOp("+",IntLit(3),BoolLit(True)) |  |
|  | BinOp("\*",BinOp("+",IntLit(3),FloatLit(3.4)),BinOp(">",IntLit(3),FloatLit(2.1))) | Type Mismatch In Expression: BinOp(">",IntLit(3),FloatLit(2.1)) | Type Mismatch In Expression: BinOp(">",IntLit(3),FloatLit(2.1)) |  |
|  | BinOp("&&",BinOp(">",BinOp("-",IntLit(3),FloatLit(3.4)),UnOp("-",FloatLit(2.1))),UnOp("-",BoolLit(True))) | Type Mismatch In Expression: UnOp("-",BoolLit(True)) | Type Mismatch In Expression: UnOp("-",BoolLit(True)) |  |
|  | UnOp("-",BinOp(">",BinOp("-",IntLit(3),FloatLit(3.4)),UnOp("-",FloatLit(2.1)))) | Type Mismatch In Expression: UnOp("-",BinOp(">",BinOp("-",IntLit(3),FloatLit(3.4)),UnOp("-",FloatLit(2.1)))) | Type Mismatch In Expression: UnOp("-",BinOp(">",BinOp("-",IntLit(3),FloatLit(3.4)),UnOp("-",FloatLit(2.1)))) |  |
|  | BinOp(">",BinOp("&&",BoolLit(True),BoolLit(False)),BinOp("||",BoolLit(True),UnOp("-",FloatLit(2.3)))) | Type Mismatch In Expression: BinOp("||",BoolLit(True),UnOp("-",FloatLit(2.3))) | Type Mismatch In Expression: BinOp("||",BoolLit(True),UnOp("-",FloatLit(2.3))) |  |
|  | UnOp("!",BinOp("==",IntLit(3),BinOp("\*",IntLit(5),IntLit(7)))) |  |  |  |
|  | UnOp("!",BinOp("==",IntLit(3),BinOp("/",IntLit(5),IntLit(7)))) | Type Mismatch In Expression: BinOp("==",IntLit(3),BinOp("/",IntLit(5),IntLit(7))) | Type Mismatch In Expression: BinOp("==",IntLit(3),BinOp("/",IntLit(5),IntLit(7))) |  |
|  | UnOp("!",BinOp("-",IntLit(3),BinOp("/",IntLit(5),IntLit(7)))) | Type Mismatch In Expression: UnOp("!",BinOp("-",IntLit(3),BinOp("/",IntLit(5),IntLit(7)))) | Type Mismatch In Expression: UnOp("!",BinOp("-",IntLit(3),BinOp("/",IntLit(5),IntLit(7)))) |  |
|  | BinOp("/",IntLit(8),BinOp("<",IntLit(3),IntLit(8))) | Type Mismatch In Expression: BinOp("/",IntLit(8),BinOp("<",IntLit(3),IntLit(8))) | Type Mismatch In Expression: BinOp("/",IntLit(8),BinOp("<",IntLit(3),IntLit(8))) |  |
|  | BinOp("||",BoolLit(True),BinOp("<",IntLit(3),IntLit(8))) |  |  |  |

Passed all tests!

**Correct**

Marks for this submission: 1.00/1.00.

### **Question**2

Correct

Mark 1.00 out of 1.00

Flag question

#### **Question text**

Given the AST declarations as follows:

class Program: #decl:List[VarDecl],exp:Exp

class VarDecl: #name:str,typ:Type

class Type(ABC): #abstract class

class IntType(Type)

class FloatType(Type)

class BoolType(Type)

class Exp(ABC): #abstract class

class BinOp(Exp): #op:str,e1:Exp,e2:Exp #op is +,-,\*,/,&&,||, >, <, ==, or  !=

class UnOp(Exp): #op:str,e:Exp #op is -, !

class IntLit(Exp): #val:int

class FloatLit(Exp): #val:float

class BoolLit(Exp): #val:bool

class Id(Exp): #name:str

and the Visitor class is declared as follows:

class StaticCheck(Visitor):

    def visitProgram(self,ctx:Program,o):pass

    def visitVarDecl(self,ctx:VarDecl,o): pass

    def visitIntType(self,ctx:IntType,o):pass

    def visitFloatType(self,ctx:FloatType,o):pass

    def visitBoolType(self,ctx:BoolType,o):pass

    def visitBinOp(self,ctx:BinOp,o): pass

    def visitUnOp(self,ctx:UnOp,o):pass

    def visitIntLit(self,ctx:IntLit,o): pass

    def visitFloatLit(self,ctx,o): pass

    def visitBoolLit(self,ctx,o): pass

    def visitId(self,ctx,o): pass

Rewrite the body of the methods in class StaticCheck to check the following type constraints:

* + , - and \* accept their operands in int or float type and return float type if at least one of their operands is in float type, otherwise, return int type
* / accepts their operands in int or float type and returns float type
* !, && and || accept their operands in bool type and return bool type
* >, <, == and != accept their operands in any type but must in the same type and return bool type
* the type of an Id is from the declarations, if the Id is not in the declarations, exception UndeclaredIdentifier should be raised with the name of the Id.

If the expression does not conform the type constraints, the StaticCheck will raise exception TypeMismatchInExpression with the innermost sub-expression that contains type mismatch.

Your code starts at line 90

**For example:**

| **Test** | **Result** |
| --- | --- |
| Program([VarDecl("x",IntType())],BinOp("\*",BinOp("+",Id("x"),FloatLit(3.4)),BinOp(">",IntLit(3),FloatLit(2.1)))) | Type Mismatch In Expression: BinOp(">",IntLit(3),FloatLit(2.1)) |

Answer:(penalty regime: 0 %)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

class StaticCheck(Visitor):

def visitProgram(self,ctx:Program,o):

self.visit(ctx.exp,ctx.decl)

def visitVarDecl(self,ctx:VarDecl,o): pass

def visitBinOp(self,ctx:BinOp,o):

t1 = self.visit(ctx.e1,o)

t2 = self.visit(ctx.e2,o)

if ctx.op in ['+','-','\*']:

if t1 == 3 or t2 == 3:

raise TypeMismatchInExpression(ctx)

if t1 == 2 or t2 == 2:

return 2

else:

return 1

if ctx.op == '/':

if t1 == 3 or t2 == 3:

raise TypeMismatchInExpression(ctx)

return 2

if ctx.op == '&&' or ctx.op == '||':

if t1 != 3 or t2 != 3:

raise TypeMismatchInExpression(ctx)

return 3

if ctx.op in ['>','<','==','!=']:

if t1 != t2:

raise TypeMismatchInExpression(ctx)

return 3

def visitUnOp(self,ctx:UnOp,o):

t0 = self.visit(ctx.e,o)

if ctx.op == '-':

if t0 == 3:

raise TypeMismatchInExpression(ctx)

return t0

if ctx.op == '!':

if t0 != 3:

raise TypeMismatchInExpression(ctx)

return t0

def visitIntLit(self,ctx:IntLit,o):

return 1

def visitFloatLit(self,ctx,o):

return 2

def visitBoolLit(self,ctx,o):

return 3

def visitId(self,ctx,o):

for var in o:

if ctx.name == var.name:

if type(var.typ) is IntType:

return 1

if type(var.typ) is FloatType:

return 2

if type(var.typ) is BoolType:

return 3

raise UndeclaredIdentifier(ctx.name)

#### **Feedback**

|  | **Test** | **Expected** | **Got** |  |
| --- | --- | --- | --- | --- |
|  | Program([],BinOp("+",IntLit(3),BoolLit(True))) | Type Mismatch In Expression: BinOp("+",IntLit(3),BoolLit(True)) | Type Mismatch In Expression: BinOp("+",IntLit(3),BoolLit(True)) |  |
|  | Program([VarDecl("x",IntType())],BinOp("\*",BinOp("+",Id("x"),FloatLit(3.4)),BinOp(">",IntLit(3),FloatLit(2.1)))) | Type Mismatch In Expression: BinOp(">",IntLit(3),FloatLit(2.1)) | Type Mismatch In Expression: BinOp(">",IntLit(3),FloatLit(2.1)) |  |
|  | Program([VarDecl("x",IntType()),VarDecl("y",BoolType())],BinOp("&&",BinOp(">",BinOp("-",IntLit(3),FloatLit(3.4)),UnOp("-",FloatLit(2.1))),UnOp("-",Id("y")))) | Type Mismatch In Expression: UnOp("-",Id("y")) | Type Mismatch In Expression: UnOp("-",Id("y")) |  |
|  | Program([VarDecl("x",IntType())],UnOp("-",BinOp(">",BinOp("-",Id("x"),FloatLit(3.4)),UnOp("-",FloatLit(2.1))))) | Type Mismatch In Expression: UnOp("-",BinOp(">",BinOp("-",Id("x"),FloatLit(3.4)),UnOp("-",FloatLit(2.1)))) | Type Mismatch In Expression: UnOp("-",BinOp(">",BinOp("-",Id("x"),FloatLit(3.4)),UnOp("-",FloatLit(2.1)))) |  |
|  | Program([VarDecl("x",BoolType()),VarDecl("y",BoolType()),VarDecl("z",FloatType())],BinOp(">",BinOp("&&",Id("x"),Id("y")),BinOp("||",BoolLit(True),UnOp("-",Id("z"))))) | Type Mismatch In Expression: BinOp("||",BoolLit(True),UnOp("-",Id("z"))) | Type Mismatch In Expression: BinOp("||",BoolLit(True),UnOp("-",Id("z"))) |  |
|  | Program([VarDecl("x",IntType()),VarDecl("y",IntType()),VarDecl("z",IntType())],UnOp("!",BinOp("==",Id("z"),BinOp("\*",Id("y"),Id("x"))))) |  |  |  |
|  | Program([VarDecl("x",IntType()),VarDecl("y",IntType()),VarDecl("z",IntType())],UnOp("!",BinOp("==",Id("x"),BinOp("/",Id("y"),Id("z"))))) | Type Mismatch In Expression: BinOp("==",Id("x"),BinOp("/",Id("y"),Id("z"))) | Type Mismatch In Expression: BinOp("==",Id("x"),BinOp("/",Id("y"),Id("z"))) |  |
|  | Program([VarDecl("x",IntType()),VarDecl("y",IntType()),VarDecl("z",IntType())],UnOp("!",BinOp("-",Id("z"),BinOp("/",Id("y"),Id("x"))))) | Type Mismatch In Expression: UnOp("!",BinOp("-",Id("z"),BinOp("/",Id("y"),Id("x")))) | Type Mismatch In Expression: UnOp("!",BinOp("-",Id("z"),BinOp("/",Id("y"),Id("x")))) |  |
|  | Program([VarDecl("x",IntType()),VarDecl("y",IntType()),VarDecl("z",IntType())],BinOp("/",Id("x"),BinOp("<",Id("y"),Id("z")))) | Type Mismatch In Expression: BinOp("/",Id("x"),BinOp("<",Id("y"),Id("z"))) | Type Mismatch In Expression: BinOp("/",Id("x"),BinOp("<",Id("y"),Id("z"))) |  |
|  | Program([VarDecl("x",IntType()),VarDecl("y",IntType()),VarDecl("z",IntType())],BinOp("||",BoolLit(True),BinOp("<",IntLit(3),Id("t")))) | Undeclared Identifier: t | Undeclared Identifier: t |  |

Passed all tests!

**Correct**

Marks for this submission: 1.00/1.00.

### **Question**3

Correct

Mark 1.00 out of 1.00

Flag question

#### **Question text**

Given the AST declarations as follows:

class Program: #decl:List[VarDecl],stmts:List[Assign]

class VarDecl: #name:str

class Assign: #lhs:Id,rhs:Exp

class Exp(ABC): #abstract class

class BinOp(Exp): #op:str,e1:Exp,e2:Exp #op is +,-,\*,/,+.,-.,\*.,/., &&,||, >, >., >b, =, =., =b

class UnOp(Exp): #op:str,e:Exp #op is -,-., !,i2f, floor

class IntLit(Exp): #val:int

class FloatLit(Exp): #val:float

class BoolLit(Exp): #val:bool

class Id(Exp): #name:str

and the Visitor class is declared as follows:

class StaticCheck(Visitor):

    def visitProgram(self,ctx:Program,o):pass

    def visitVarDecl(self,ctx:VarDecl,o): pass

    def visitAssign(self,ctx:Assign,o): pass

    def visitBinOp(self,ctx:BinOp,o): pass

    def visitUnOp(self,ctx:UnOp,o):pass

    def visitIntLit(self,ctx:IntLit,o): pass

    def visitFloatLit(self,ctx,o): pass

    def visitBoolLit(self,ctx,o): pass

    def visitId(self,ctx,o): pass

Rewrite the body of the methods in class StaticCheck to infer the type of identifiers and check the following type constraints:

* + , - , \*, / accept their operands in int type and return int type
* +., -., \*., /. accept their operands in float type and return float type
* > and = accept their operands in int type and return bool type
* >. and =. accept their operands in float type and return bool type
* !, &&, ||, >b and =b accept their operands in bool type and return bool type
* i2f accepts its operand in int type and return float type
* floor accept its operand in float type and return int type
* In an assignment statement, the type of lhs must be the same as that of rhs, otherwise, the exception TypeMismatchInStatement should be raised together with the assignment statement.
* the type of an Id is inferred from the above constraints in the first usage,
  + if the Id is not in the declarations, exception UndeclaredIdentifier should be raised together with the name of the Id, or
  + If the Id cannot be inferred in the first usage, exception TypeCannotBeInferred should be raised together with the name of the assignment statement.
* If an expression does not conform the type constraints, the StaticCheck will raise exception TypeMismatchInExpression with the expression.

Your code starts at line 95

Answer:(penalty regime: 0 %)

class StaticCheck(Visitor):

def visitProgram(self,ctx:Program,o):

o = {}

for decl in ctx.decl:

self.visit(decl,o)

for stmt in ctx.stmts:

self.visit(stmt,o)

def visitVarDecl(self,ctx:VarDecl,o):

o[ctx.name] = "none"

def visitAssign(self,ctx:Assign,o):

rhs = self.visit(ctx.rhs,o)

lhs = self.visit(ctx.lhs, o)

if lhs == "none":

if rhs == "none":

raise TypeCannotBeInferred(ctx)

else:

o[ctx.lhs.name] = rhs

lhs = rhs

if rhs == "none":

if lhs == "none":

raise TypeCannotBeInferred(ctx)

o[ctx.rhs.name] = lhs

rhs=lhs

if lhs != rhs:

raise TypeMismatchInStatement(ctx)

def visitBinOp(self,ctx:BinOp,o):

ltype = self.visit(ctx.e1,o)

rtype = self.visit(ctx.e2,o)

if ctx.op in ["+", "-", "\*", "/"]:

if ltype == "none":

o[ctx.e1.name] = "int"

ltype = "int"

if rtype == "none":

o[ctx.e2.name] = "int"

rtype = "int"

if ltype == "int" and rtype == "int":

return "int"

raise TypeMismatchInExpression(ctx)

if ctx.op in ["+.", "-.", "\*.", "/."]:

if ltype == "none":

o[ctx.e1.name] = "float"

ltype = "float"

if rtype == "none":

o[ctx.e2.name] = "float"

rtype = "float"

if ltype == "float" and rtype == "float":

return "float"

raise TypeMismatchInExpression(ctx)

if ctx.op in [">","="]:

if ltype == "none":

o[ctx.e1.name] = "int"

ltype = "int"

if rtype == "none":

o[ctx.e2.name] = "int"

rtype = "int"

if ltype == "int" and rtype == "int":

return "bool"

raise TypeMismatchInExpression(ctx)

if ctx.op in [">.", "=."]:

if ltype == "none":

o[ctx.e1.name] = "float"

ltype = "float"

if rtype == "none":

o[ctx.e2.name] = "float"

rtype = "float"

if ltype == "float" and rtype == "float":

return "bool"

raise TypeMismatchInExpression(ctx)

if ctx.op in ["&&", "||", ">b", "=b"]:

if ltype == "none":

o[ctx.e1.name] = "bool"

ltype = "bool"

if rtype == "none":

o[ctx.e2.name] = "bool"

rtype = "bool"

if ltype == "bool" and rtype == "bool":

return "bool"

raise TypeMismatchInExpression(ctx)

def visitUnOp(self,ctx:UnOp,o):

typ = self.visit(ctx.e,o)

if ctx.op == "-":

if typ == "none":

o[ctx.e.name] = "int"

typ = "int"

if typ == "int":

return "int"

raise TypeMismatchInExpression(ctx)

if ctx.op == "-.":

if typ == "none":

o[ctx.e.name] = "float"

typ = "float"

if typ == "float":

return "float"

raise TypeMismatchInExpression(ctx)

if ctx.op == "!":

if typ == "none":

o[ctx.e.name] = "bool"

typ = "bool"

if typ == "bool":

return "bool"

raise TypeMismatchInExpression(ctx)

if ctx.op == "i2f":

if typ == "none":

o[ctx.e.name] = "int"

typ = "int"

if typ == "int":

return "float"

raise TypeMismatchInExpression(ctx)

if ctx.op == "floor":

if typ == "none":

o[ctx.e.name] = "float"

typ = "float"

if typ == "float":

return "int"

raise TypeMismatchInExpression(ctx)

def visitIntLit(self,ctx:IntLit,o):

return "int"

def visitFloatLit(self,ctx,o):

return "float"

def visitBoolLit(self,ctx,o):

return "bool"

def visitId(self,ctx,o):

if ctx.name in o:

return o[ctx.name]

raise UndeclaredIdentifier(ctx.name)

#### **Feedback**

|  | **Test** | **Expected** | **Got** |  |
| --- | --- | --- | --- | --- |
|  | Program([VarDecl("x")],[Assign(Id("x"),BinOp("+",IntLit(3),BoolLit(True)))]) | Type Mismatch In Expression: BinOp("+",IntLit(3),BoolLit(True)) | Type Mismatch In Expression: BinOp("+",IntLit(3),BoolLit(True)) |  |
|  | Program([VarDecl("x")],[Assign(Id("x"),BinOp("\*",BinOp("+",Id("x"),IntLit(3.4)),BinOp("-",Id("x"),FloatLit(2.1))))]) | Type Mismatch In Expression: BinOp("-",Id("x"),FloatLit(2.1)) | Type Mismatch In Expression: BinOp("-",Id("x"),FloatLit(2.1)) |  |
|  | Program([VarDecl("x"),VarDecl("y"),VarDecl("z")],[Assign(Id("z"),BinOp("&&",BinOp(">",BinOp("-",Id("x"),IntLit(3)),UnOp("-",Id("y"))),UnOp("!",Id("y"))))]) | Type Mismatch In Expression: UnOp("!",Id("y")) | Type Mismatch In Expression: UnOp("!",Id("y")) |  |
|  | Program([VarDecl("x")],[Assign(Id("x"),UnOp("-",BinOp(">.",BinOp("-.",Id("x"),FloatLit(3.4)),UnOp("-.",FloatLit(2.1)))))]) | Type Mismatch In Expression: UnOp("-",BinOp(">.",BinOp("-.",Id("x"),FloatLit(3.4)),UnOp("-.",FloatLit(2.1)))) | Type Mismatch In Expression: UnOp("-",BinOp(">.",BinOp("-.",Id("x"),FloatLit(3.4)),UnOp("-.",FloatLit(2.1)))) |  |
|  | Program([VarDecl("x"),VarDecl("y"),VarDecl("z")],[Assign(Id("x"),BinOp(">b",BinOp("&&",Id("x"),Id("y")),BinOp("||",BoolLit(False),BinOp(">",Id("z"),IntLit(3))))),Assign(Id("z"),Id("x"))]) | Type Mismatch In Statement: Assign(Id("z"),Id("x")) | Type Mismatch In Statement: Assign(Id("z"),Id("x")) |  |
|  | Program([VarDecl("x"),VarDecl("y"),VarDecl("z")],[Assign(Id("x"),UnOp("!",BinOp("=",Id("z"),BinOp("\*",Id("y"),Id("x")))))]) | Type Mismatch In Statement: Assign(Id("x"),UnOp("!",BinOp("=",Id("z"),BinOp("\*",Id("y"),Id("x"))))) | Type Mismatch In Statement: Assign(Id("x"),UnOp("!",BinOp("=",Id("z"),BinOp("\*",Id("y"),Id("x"))))) |  |
|  | Program([VarDecl("x"),VarDecl("y")],[Assign(Id("x"),Id("y"))]) | Type Cannot Be Inferred: Assign(Id("x"),Id("y")) | Type Cannot Be Inferred: Assign(Id("x"),Id("y")) |  |
|  | Program([VarDecl("x"),VarDecl("y"),VarDecl("z")],[Assign(Id("x"),UnOp("-.",BinOp("-.",Id("z"),BinOp("/.",UnOp("i2f",Id("y")),Id("x"))))),Assign(Id("y"),FloatLit(3.2))]) | Type Mismatch In Statement: Assign(Id("y"),FloatLit(3.2)) | Type Mismatch In Statement: Assign(Id("y"),FloatLit(3.2)) |  |
|  | Program([VarDecl("x"),VarDecl("y"),VarDecl("z")],[Assign(Id("z"),IntLit(3)),Assign(Id("x"),Id("z")),Assign(Id("y"),BinOp("&&",Id("x"),BinOp("=b",Id("y"),BoolLit(True))))]) | Type Mismatch In Expression: BinOp("&&",Id("x"),BinOp("=b",Id("y"),BoolLit(True))) | Type Mismatch In Expression: BinOp("&&",Id("x"),BinOp("=b",Id("y"),BoolLit(True))) |  |
|  | Program([VarDecl("x"),VarDecl("y"),VarDecl("z")],[Assign(Id("t"),BinOp("||",BoolLit(True),BinOp(">",IntLit(3),Id("z"))))]) | Undeclared Identifier: t | Undeclared Identifier: t |  |
|  | Program([VarDecl("x"),VarDecl("y"),VarDecl("z")],[Assign(Id("x"),FloatLit(3.0)),Assign(Id("x"),Id("y")),Assign(Id("z"),BinOp(">",IntLit(3),Id("y")))]) | Type Mismatch In Expression: BinOp(">",IntLit(3),Id("y")) | Type Mismatch In Expression: BinOp(">",IntLit(3),Id("y")) |  |

Passed all tests!

**Correct**

Marks for this submission: 1.00/1.00.

Bottom of Form