Dicussion Of Problems

Jump Commands

To solve this problem we need to implement to commands, a jump command and a conditional jump command. The JumpC command constructor takes a single integer as input, so simply when the jump command is executed the current position in the program command list is set to the integer contained in the jump command.

The JumpC command constructor boolean value and a integer. When the JumpC command is executed if the value on top of the stack is equal to the given boolean value then we jump to the given integer.

As we translate the program we give the jump commands relitive locations. Then after the translation is complete we do a second pass through the list of instructions assigning each instruction an ID and updating the relitive jumps to absolute jumps.

Part 2 Type Checking

We created an abstract data type with two types Integer and Boolean, this is used when type checking to compare the types of two things. We also updated the val data type to have two parts, I and B .

We where given four things we had to ensure so we dicuss them individually here.

1. **No variable may be declared twice:** This is simple enough to check we just look at the list of declarations and ensure that all the variable names are uniquic.
2. **Every variable used in the program must be declared:** While type checking if we encouter a variable that hasnt been declared then when we look up its type and we cant find it in the declrations we can throw an error
3. **Operators must be applyed to arguments of the correct type:** We create two functions, typeopin maps a binary operation to the type of input it accepts and typeopreturn maps a binary operation to the type of output it gives. So to make sure that a use of Plus is allowed we make sure the type of the given expressions is the same as (typeopin Plus). However we have a problem, An operation like Eq can take integer or boolean values. To solve this we add another type 'Any' which is only used when taking about binary op inputs.

4. **Variables must only be assigned values of there declared type:** When checking an assignment we first lookup the type of the variable, then we check the type of the expression we are trying to assign to the variable if the two types are equal then we allow it, otherwise we throw an error.

Another thing we had to think about was in if statments and while loops the expression given as the condition had to evaluate to a boolean.

Simple Procedures

A Call statment was created to handle calling a procedure, so far it just takes a name `Call <name>`. There where two issues to think about here

1. **Ensuring a meaningful program:** Firstly we must ensure that for any call command the name of the given procedure is a valid name. We also must ensure that all statments contained in a procedure must be valid. We also want to check to make sure all the procedure names are uniquic.
2. **Implementing procedure calling and return:** When we translate the procedures we construct a list of tuples of procedure name and prodecure starting position in the code. We also create three new commands 'JumpP <procedure name>', 'JumpS' and 'StoreL <offset>'. When a JumpP command is executed the starting position is looked up in the list of tuples mentioned before, we then move to that position in the instructions. The StoreL puts the current position in the instructions plus some offset on to the stack (we need the offset because we dont want to return to the same place we want to return to after the procedure call) and finally JumpS takes the top element on the stack and jumps to that position in the list of instructions.

Part 4

First we need to think about scoping, insted of having a single store we will have a stack of stores. Every time we call a procedure we create a new store and put it on the stack of stores. We also initilize all these stores to contain default values for the declared variables (and paramaters if its a procedure call) i.e. 0for integers and false for booleans.

When we are reading a value from the store we decend thouguh our stores untill we find the first place the variable we are looking for contained in a store then we read that value. Likewise when writing to the store we decend through the stores until we find the first place the variable is mentioned. This allows for masking global variables by declaring them in a procedure.

To add paramaters during execution we first put all paramater values onto the stack. Then the first instructions of a translated procedure read the values back of the stack (in reverse order otherwise we wont have the correct variable with the correct value) and save then save to the store. Sinse we set up default values these paramaters will be saved to the top most store.

Unfourtunatly we cant peform type checking on the paramaters being used in a Call statment because of the way the scope works we dont know what paramaters will be available at run time.

Code

```
module Straight where
import Data.List as L
```

Compiler and interpreter for simple straight line programs.

A straight line program is just a list of assignment statements, where an expression can contain variables, integer constants and arithmetic operators.

```
data Prog = Prog [(Var, Type)] [Procedure] [Stmt]
            deriving (Show)
```

Abstract type declration for a procedure. Takes a name, list of paramaters, list of local variables and then a list of statments.

```
data Procedure = Procedure String [(Var, Type)] [(Var, Type)] [Stmt] deriving (Show)

data Stmt = Asgn Var Exp
            | Call String [Var]
            | If Exp [Stmt] [Stmt]
            | While Exp [Stmt]
            deriving (Show)

data Exp = Const Val
         | Var Char
         | Bin Op Exp Exp
           deriving (Show)

data Op = Plus | Minus | Times | Div | Eq | Ne | Lt | Le | Gt | Ge | And | Or
          deriving (Show)
```

The store is a list of variable names and their values. For now, a variable now is just a single character and a value is an integer.

```
type Store = [(Var, Val)]

type Var = Char

data Type = Integer | Boolean | Any deriving (Show, Eq)

data Val = I Int | B Bool deriving (Show, Eq)
```

Straight line programs are translated into code for a simple stack-oriented virtual machine.

A VM code program is a list of commands, where a command is either a load immdiate (which loads a constant onto the stack), load (which loads the value of a variable onto the stack), sstore (which saves the value at the top of the stack into memory, and deletes it), or an arithmetic operation (which applies an operations to the two values at the top of the stack and replaces them by the result). We have three Jump commands, a regular jump whichs simply takes us to the given position in the instruction list. a JumpC is also supplied a value, the conditional jump command only jumps to the given position if the value on top of the stack is the same as the given value. JumpP command jumps to the starting position of a procedure. JumpS takes the top value of the top of the stack and jumps to that position in the instructions. StoreL takes an integer, when executed stores the current position in the instructions plus the given integer to the stack.

```
type Code = [Command]

data Command = LoadI Val | Load Var | Store Var | BinOp Op |
               Jump Int | JumpC Val Int | JumpP String | JumpS | StoreL Int
               deriving (Show)

type Stack = [Val]
```

Run a program, by compiling and then executing the resulting code The program is run with an initially empty store and empty stack, and the output is just the resulting store. This could be modified to provide an initial store.

```
run :: Prog -> [Store]
```

Before we run the program we must run checks to veryfi as much of the program as we can and then translate the program

```
run prog@(Prog var pro stmt)
  | ok == False = error "There is a type error in your program"
  | otherwise = snd (execute code proc ([], [(defaultstore var)]))
```

Here we get the translated code along with a list of tripples each one containing a procedure name, where in the instructions the procedure starts and the list of variables used in that procedure so we can initialse the default store

```
  where (proc, code) = translate prog
```

This is the top level of the type checking, ok must be true for the program to run. How ever we should never get ok as False, all the type checking functions throw an error insted of returning False.

```
        ok = t && d && p
        t = typecheck prog
        d = vardeclarationcheck var
        p = proceduredeclarationcheck pro
```

vardeclarationcheck takes a list of varaible declarations and ensures that they are all uniquic, it dosnt make sense to be able to define 'b' as an Integer and as a Boolean. Works by getting a uniquic list of variable names and checks to see if the size of this is equal to the origonal type declaration list.

```
vardeclarationcheck :: [(Var, Type)] -> Bool
vardeclarationcheck d = length (unqvar d []) == length (d)
```

unqvar returns a list containing only uniquc variable names, if we encounter a duplicate we throw an error.

```
  where unqvar [] e = e
        unqvar ((a, b):ds) e
          | (a `elem` e) = error "Multiple variable declaration error"
          | otherwise = unqvar ds (a:e)
```

proceduredeclarationcheck is given a list of procedures and ensures that there are no two procedures with the same name. Works the same way as vardeclarationcheck. Constructs a uniquic list of names and checks the size.

```
proceduredeclarationcheck :: [Procedure] -> Bool
proceduredeclarationcheck p = length (unqproc p []) == length (p)
```

unqproc constructs a uniqic list of procedure names and throws an error if we encounter a duplicate

```
  where unqproc [] e = e
        unqproc ((Procedure name _ _ _):ps) e
          | (name `elem` e) = error "Multiple procedures with same name"
          | otherwise = unqproc ps (name:e)
```

Given a program ensures that there are no type errors. By checking both all the statments in the program and also checking all the procedures in the program.

```
typecheck :: Prog -> Bool
typecheck (Prog d p s) = ok
  where ok = sok && pok
        sok = typecheckstmts s p d
        pok = procedurecheck p p d
```

procedurecheck takes a list of procedures and a list of the variables which have been declared globally and ensures that all operations and assignments make sense with regrads to the types of variables and functions.

```
procedurecheck :: [Procedure] -> [Procedure] -> [(Var, Type)] -> Bool
```

There are no type errors in an empty list of procedures so we can return true

```
procedurecheck [] _ _ = True
procedurecheck ((Procedure n p v s):xs) proc d
```

It dosnt make sense for us to be able to declare a paramater with the same name as a local varialbe so we must ensure that the two lists are disjoint.

```
  | length common /= 0 = error "Paramaters and variables should be disjoint"
```

Otherwise we ensure that all the statments are correct, and rather than just allowing the global variables we want to allow global + local.

```
  | otherwise = (typecheckstmts s proc vard) && (procedurecheck xs proc d)
```

common is the intersection by variable name beween the paramaters and the local variables

```
    where common = L.intersectBy (\(n1, _) (n2, _) -> n1 == n2) p v
```

vard is the union by variable name. The order of the lists is important here we want to have (p ++ v) first as anything in here takes priority over anything global. This allows for variable shadowing.

```
          vard = L.unionBy (\(n1, _) (n2, _) -> n1 == n2) (p ++ v) d
```

typecheckstmts takes a list of statements and a list of the allowed variables with there types. And ensures that all the statments are valid

```
typecheckstmts :: [Stmt] -> [Procedure] -> [(Var, Type)] -> Bool
typecheckstmts [] _ _ = True
typecheckstmts (x:xs) p d = (typecheck' x p d) && (typecheckstmts xs p d)
```

typecheck' is where we define what makes each of the different statments valid

```
typecheck' :: Stmt -> [Procedure] -> [(Var, Type)] -> Bool
```

for an assignment to be valid the type of the expression must be the same as the type of the variable

```
typecheck' (Asgn v e) _ d
  | (typeexp e d) == (typelookup v d) = True
  | otherwise = error "Type mismatch error with Asgn statment"
```

an if statment is valid if the expression will evaluate to a boolean and all the statments in both the true and false branches are valid.

```
typecheck' (If e t f) p d
  | ((typeexp e d) == Boolean && (typecheckstmts t p d) && (typecheckstmts f p d)) = True
  | otherwise = error "Type mismatch error with If statment"
```

a while loop is valid only when the expression evaluates to a boolean and all of the statments are also valid

```
typecheck' (While e s) p d
  | ((typeexp e d) == Boolean && (typecheckstmts s p d)) = True
  | otherwise = error "Type mismatch error with While statment"
```

We always evaluate a call to be true as there is no way to check this before run time, we dont know where it will of been called from and what variables are available.

```
typecheck' (Call name _) p d = name == n
  where (Procedure n _ _ _) = procedurelookup name p
```

procedurelookup takes a procedure name and a list of procedures. And searches the list of procedures for one with the given name. Throws an error if such a procedure dosnt exsist.

```
procedurelookup :: String -> [Procedure] -> Procedure
procedurelookup name [] = error "Couldent find procedure"
procedurelookup name (a@(Procedure n _ _ _):ps)
  | n == name = a
  | otherwise = procedurelookup name ps
```

typelookup takes a variable name and a list of variable declarations. It looks through the list of declarations until we reach one with the given name, then we return the asociated type. Throws an error if we cant find a declration with the given name.

```
typelookup :: Var -> [(Var, Type)] -> Type
typelookup v d = getType v d
  where getType v [] = error "Variable dosnt exsist"
        getType v ((n, t):ds)
          | v == n = t
          | otherwise = getType v ds
```

typeexp takes an expression and a list of variable declarations. It returns the type of the given expression

```
typeexp :: Exp -> [(Var, Type)] -> Type
```

Clearly the type of a constant integer is Integer and for a constant boolean is Boolean

```
typeexp (Const (I _)) d = Integer
typeexp (Const (B _)) d = Boolean
```

To find the type of a variable we simply look it up in the declaration list

```
typeexp (Var a) d = typelookup a d
```

For a binary operation first we ensure that the types of the input to the operation are valid. Then if they are we return the output type of that operation. If the input types do not match we throw an error

```
typeexp (Bin o e1 e2) d
  | ((optype == e1type) && (optype == e2type))
    || (e1type == e2type && (typeopin o) == Any) = (typeopreturn o)
  | otherwise = error "Type mismatch error with binary operation"
  where optype = typeopin o
        e1type = typeexp e1 d
        e2type = typeexp e2 d
```

typeopreturn simply defines the return types of each of the binary operations

```
typeopreturn :: Op -> Type
typeopreturn Plus = Integer
typeopreturn Minus = Integer
typeopreturn Times = Integer
typeopreturn Div = Integer
typeopreturn Lt = Boolean
typeopreturn Le = Boolean
typeopreturn Gt = Boolean
typeopreturn Ge = Boolean
typeopreturn Eq = Boolean
typeopreturn Ne = Boolean
typeopreturn And = Boolean
typeopreturn Or = Boolean
```

typeopin defines the accepible input types for each of the binary operations

```
typeopin :: Op -> Type
typeopin Plus = Integer
typeopin Minus = Integer
typeopin Times = Integer
typeopin Div = Integer
typeopin Lt = Integer
typeopin Le = Integer
typeopin Gt = Integer
typeopin Ge = Integer
typeopin Eq = Any
typeopin Ne = Any
typeopin And = Boolean
typeopin Or = Boolean
```

Translate straight line program into stack machine code

```
addIds :: [Command] -> [(Command, Int)]
addIds x = (addIds' 0 x)
```

Takes a translated program and assigns an id to each of the instructions, also updaes the Jump and JumpC commands so the positions are absolute insted of relitive.

```
addIds' :: Int -> [Command] -> [(Command, Int)]
addIds' n ((Jump i):xs) = ((Jump (n+i)), n):(addIds' (n+1) xs)
addIds' n ((JumpC a b):xs) = (JumpC a (n+b), n):(addIds' (n+1) xs)
addIds' n (x:xs) = (x, n):(addIds' (n+1) xs)
addIds' n [] = []
```

translate takes a program and a tuple containing two things, firstly a list of tripples representing a procedure. Each of these tripples contains a procedure name, a starting position in the code and the list of variables used in the procedure. Secondly we havve the translated program.

```
translate :: Prog -> ([(String, Int, [(Var, Type)])], [(Command, Int)])

translate (Prog d p stmts) = (procnames, addIds (proccode ++ (trans stmts)))
                             where (procnames, proccode) = transprocedures p
```

transprocedures takes a list of procedures and returns a tuple containing the list of tripples as dicussed above and the code for all the translated procedures.

```
transprocedures :: [Procedure] -> ([(String, Int, [(Var, Type)])], Code)
```

We add a Jump command before the procedure code so the only way to execute a procedure is to call it, we cant just accerdentally execute one. Because we add this extra instruction we have to shift the starting position of all the procedures by 1.

```
transprocedures p = (shift names, (Jump (length(code) + 1)):code)
                    where (names, code) = (transprocedures' p 0)
                          shift [] = []
                          shift ((name, i, v):xs) = (name, i+1, v):(shift xs)
```

transprocedures' takes a list of procedures and a number. This number is counting the number of instructions so far so we know the starting position of each of the procedures.

```
transprocedures' :: [Procedure] -> Int -> ([(String, Int, [(Var, Type)])], Code)

transprocedures' [] _ = ([], [])

transprocedures' ((Procedure n p v s):ps) l = ((n, l, (p ++ v)):names, ((reverse lp) ++ stat
                                          where (names, code) = transprocedures' ps (l + (leng
```

translate the statments in the procedure

```
                                                statments = trans s
```

Create a list of instructions to store the params given to the procedure

```
                                                    lp = loadparams p
```

loadparams function takes a list of paramaters and returns a list of commands to save the paramaters to the store. We can do this as before the procedure is called all the paramaters are loaded onto the stack.

```
                                                    loadparams [] = []
                                                    loadparams ((p, _):ps) = (Store p):(loadparams
```

trans takes a list of statments and converts them to a list of commands.

```
trans :: [Stmt] -> Code
trans [] = []
trans (stmt : stmts) = (trans' stmt) ++ (trans stmts)
```

trans' is where we define how to translate each of the different types of statment

```
trans' :: Stmt -> Code
```

If we want to store an expression to a variable first we must execute the code to evaluate the expression then we execute a store command to save the top of the stack to the variable we are assigning to. So to translate an assignment we first translate the expression and then add a Store command.

```
trans' (Asgn var exp) = (transexp exp) ++ [Store var]
```

To translate an if statment first we want to execute the condition expression then we have a condition jump which will execute if the value on top of the stack is False (i.e. taking us to the else branch). Then we add the translated statments for the if branch folowed by a jump to skip the else block when the if block is executed, lastly folowed by the else block statments.

```
trans' (If c t f) = conds ++ [JumpC (B False) ((length trueStatments) + 2)] ++
                    trueStatments ++ [Jump ((length falseStatments) + 1)] ++
                    falseStatments
```

translate the condition expression

```
                  where conds = transexp c
```

translate the if block statments

```
                    trueStatments = trans t
```

translate the else block statments

```
                    falseStatments = trans f
```

to translate a while loop first we translate the condition then we add a jump command to take us past the while loop instrcutions if the condition evaluates to false folowing that we have the translated loop statments

```
trans' (While c s) = conds ++ [JumpC (B False) ((length statments) + 2)] ++ statments
                           ++ [Jump (-((length statments) + 1 + (length conds)))]
                     where conds = transexp c
                           statments = trans s
```

when translating a call to a procedure first we add a StoreL command to save our location + an offset to the stack so we can return to the correct position. Then we add instructions to load the paramaters to the stack. Finally we add a JumpP command to take us to the start of the procedure.

```
trans' (Call name params) = (StoreL (2 + length loadparams)):loadparams ++ [JumpP name]
```

The function p takes a list of paramaters and returns a list of commands to load the paramaters to the stack.

```
    where loadparams = p params
          p [] = []
          p (x:xs) = (Load x):(p xs)
```

The transexp function is where we define how to translate expressions to commands.

```
transexp :: Exp -> Code
transexp (Const n) = [LoadI n]
transexp (Var v) = [Load v]
```

We translate e2 then e1 as then the values are loaded onto the stack in a logical order. Otherwise something like `Bin Lt 1 2` will be false because its checking is $2 < 1$

```
transexp (Bin op e1 e2) = transexp e2 ++ transexp e1 ++ [BinOp op]
```

Execute a stack code program

```
execute :: [(Command, Int)] -> [(String, Int, [(Var, Type)])] -> (Stack, [Store]) -> (Stack,
```

```
execute c p ss = exec c p ss 0
```

the exec function handles executing the. At each step we get the current command execute it to get the next position, stack and stores, and so on. If at any point we end up past the end of the list of instructions we know we have finished executing.

```
exec :: [(Command, Int)] -> [(String, Int, [(Var, Type)])] -> (Stack, [Store]) -> Int -> (St
```

```
exec cmds p ss n
```

if we have reached the end of execution then return the stack and store

```
  | n >= (length cmds) = ss
```

otherwise execute the next command

```
  | otherwise = exec cmds p newss next
  where cmd = cmds !! n
        (next, newss) = execinstruction cmd p ss n
```

```
execinstruction :: (Command, Int) -> [(String, Int, [(Var, Type)])] -> (Stack, [Store]) -> I
```

When executing a jump command we just return the position contained in the jump as the next position with the current stack and store.

```
execinstruction (Jump i, _) _ ss n = (i, ss)
```

Executing a conditional jump is similar to a regular except we take the top of the stack and only if the top of the stack is equal to the value in the conditional jump do we up date the current position to the one contained in the stack. Otherwise we return the current positon plus 1.

```
execinstruction (JumpC a i, _) _ (x:stack, store) n
  | a == x = (i, (stack, store))
  | otherwise = (n+1, (stack, store))
```

JumpP searchs our list of procedures for the entry with the given name then sets the next position to the procedure start inside the entry.

```
execinstruction (JumpP name, _) p (stack, stores) n = (start, (stack, ds:stores))
```

the proc command looks through our list of procedure entries for the one with the given name. Throws an error if we cant find it, however this shouldent happen as it should of been picked up during the checking.

```
    where proc [] name = error "Error looking up procedure"
          proc (a@(n, _, _):ps) name
            | n == name = a
            | otherwise = proc ps name
          (n, start, var) = proc p name
          ds = defaultstore var
```

JumpS command sets our next position to the current value on top of the stack.

```
execinstruction (JumpS, _) _ ((I x):stack, s:store) _ = (x, (stack, store))
```

Otherwise the command isnt involving a jump so we hand off to a different function.

```
execinstruction cmd _ ss n = (n+1, exec' cmd ss)
```

The defaultstore function takes a list of variable declrations and returns a store containing all the variables with there default values. For an integer there the default value is 0 and for a Boolean is False.

```
defaultstore :: [(Var, Type)] -> [(Var, Val)]
defaultstore v = map (\(var, t) -> if t == Integer then (var, I 0) else (var, B False)) v
```

the exec' function is where we define what happens when executing commands other than jump commands

```
exec' :: (Command, Int) -> (Stack, [Store]) -> (Stack, [Store])
```

When executing a load immedaite we simply put the value in the load onto the stack

```
exec' (LoadI n, i) (stack, store) = (n:stack, store)
```

When executing a load command we just have to lookup the variable and put it on the stack.

```
exec' (Load v, i) (stack, store) = (x:stack, store)
  where x = getVal v store
```

When executing a store command we take the value on top of the stack and save it to the store under the given variable.

```
exec' (Store v, i) (x:stack, store) = (stack, store')
  where store' = setVal v x store
```

To execute a binary op take the two items on top of the stack and execute the operation on them put the result back on the stack.

```
exec' (BinOp op, i)  (x:y:stack, store) = (z:stack, store)
  where z = apply op x y
```

Executing StoreL operation just takes the current position adds an offset to it and puts it on top of the stack.

```
exec' (StoreL o, i) (stack, store) = ((I (o+i)):stack, store)

exec' _ ss = ss
```

Apply an arithmetic operator

```
apply :: Op -> (Val) -> (Val) -> Val
apply Plus (I x) (I y) = I (x+y)
apply Minus (I x) (I y) = I (x-y)
apply Times (I x) (I y) = I (x*y)
apply Div (I x) (I y) = I (x `div` y)
apply Lt (I x) (I y) = B (x < y)
apply Le (I x) (I y) = B (x <= y)
apply Gt (I x) (I y) = B (x > y)
apply Ge (I x) (I y) = B (x >= y)
apply Eq x y = B (x == y)
apply Ne x y = B (x /= y)
apply And (B x) (B y) = B (x && y)
apply Or (B x) (B y) = B (x || y)
```

Look up a variable in the store. Works by looking through our stack of stores until we find one that contains the given variable, then get the asociated value and return it. If no store has the variable then we throw an error. This shouldent happen though as it should be found at check time.

```haskell
getVal :: Var -> [Store] -> Val
getVal _ [] = error "Variable not found"
getVal v (s:stores)
  | (hasVariable v s) == True = getVal' v s
  | otherwise = getVal v stores

getVal' :: Var -> Store -> Val
getVal' v s = foldr (\(u,x) r -> if u==v then x else r) (error "Variable not found") s
```

Assign a value to a variable in the store. Works by looking through our stack of stores untill we find one that contains the variable we are looking for, if we find one then deligate to another function to update the store. If none is found an error is thrown (this shouldent happen as should be caught at check time).

```haskell
setVal :: Var -> Val -> [Store] -> [Store]
setVal _ _ [] = error "Variable not found"
setVal v x (s:stores)
  | (hasVariable v s) == True = (setVal' v x s):stores
  | otherwise = s:(setVal v x stores)

setVal' :: Var -> Val -> Store -> Store
setVal' v x ((u,y):s)
  | v == u = (v,x):s
  | otherwise = (u,y):(setVal' v x s)
```

hasVariable function searches a store to see if it contains a given varaible.

```haskell
hasVariable :: Var -> Store -> Bool
hasVariable _ [] = False
hasVariable v ((a,b):r) = if v == a then True else hasVariable v r
```

Some examples for testing

Demonstraites using a if statment

```haskell
e1 = Const (I 1)
e2 = Var 'a'
e3 = Bin Plus e1 e1
e4 = Const (I 1)
s1 = Asgn 'a' e3
s2 = Asgn 'a' e2
f1 = If (Bin Eq e1 e4) [s1,s2] []
p1 = Prog [('a', Integer)] [] [f1]
p2 = Prog [('a', Integer)] [] [s1, s2]
```

Demonstraites using a while loop

```
a1 = Const (I 1)
v1 = Var 'a'
a2 = Asgn 'a' a1
a3 = Const (I 9)
w1 = While (Bin Ne v1 a3) [a4]
b1 = Bin Plus v1 a1
a4 = Asgn 'a' b1
a5 = Const (I 0)
a6 = Asgn 'a' a5
p3 = Prog [] [] [a6, w1]
```

Should fail as we are trying to add a integer with a boolean

```
x1 = Const (I 1)
x2 = Const (B True)
y1 = Bin Plus x1 x2
z1 = Asgn 'a' y1
p4 = Prog [('a', Integer)] [] [z1]
```

Should add 1 to the varaible a until its 10, demonstrates recursive procedure calling

```
o1 = Const(I 10)
o2 = Var 'a'
o3 = Const(I 1)
m2 = Bin Plus o2 o3
m3 = Asgn 'a' m2
m4 = Call "add" []
m1 = If (Bin Ne o1 o2) [m3, m4] []
n1 = Procedure "add" [] [] [m1]
m5 = Asgn 'a' (Const (I 0))
m6 = Call "add" []
m7 = Asgn 'b' (Const (I 3))
p5 = Prog [('a', Integer), ('b', Integer)] [n1, (Procedure "Add" [] [] [])] [m5, m6, m7]
```

Demonstrates calling one procedure from another

```
proxy1 = Procedure "PAdd" [('i', Integer), ('j', Integer)] [] [Call "Add" ['i', 'j']]
k2 = Var 'g'
k3 = Var 'h'
k4 = Bin Plus k2 k3
k5 = Asgn 'a' k4
```

```
k10 = Asgn 'b' (Const (B True))
k11 = If (Bin Eq (Var 'b') (Const (B True))) [k5] [Asgn 'a' (Const (I 1))]
k1 = Procedure "Add" [('g', Integer), ('h', Integer)] [('b', Boolean)] [k10, k11]
k6 = Asgn 'a' (Const (I 0))
k7 = Asgn 'b' (Const (I 7))
k8 = Asgn 'c' (Const (I 4))
k9 = Call "PAdd" ['b', 'c']
p6 = Prog [('a', Integer), ('b', Integer), ('c', Integer)] [k1, proxy1] [k6, k7, k8, k9]
```

Demonstrates how checking will fail when there are procedures with the same name

```
p7 = Prog [('a', Integer), ('b', Integer), ('c', Integer)] [k1, k1] []
```