EECS 4302 Winter 2020
Project Milestone 3 Document 3
Team: Abdul Adeshina, Abdullah Basulaib, Chidi Okongwu, Shiyi Du

Alloys object structure lend itself nicely to our class based input language

Each class is represented as an alloy object, with its attributes corresponding to the variables declared in the declaration body
For example:

```
class Account{
        declare {
                int old_amount;
                int amount;
                int a = 5;
                bool testBool = false;
        }
}
```

Will be translated into:

```
some sig Account{
        old_amount:Int,
        amount:Int,
        a:5,
        testBool:False
}
```

Assumptions are represented as facts in alloy. A fact in alloy, as hinted by the name, declares some expression as a universal fact (eg. x > 3) and thus, the verification tool need not verify cases for which x <= 3, this serves a similar domain restricting function that we defined the assumption block to serve in doc1.

One thing we do need to worry about is the scope for the variables, the variables can only be accessed within the class, so the we we simulate that in alloy is to add prefixes to variables, for example: x is translated into n.x, where n is referring to the corresponding class

For example

```
        assume{
                amount > 0;
        }
```

Is in the class Account, so it will be translated into:

```
fact AccountFact {
        all n: Account| (n.amount) > (0)
}
```

Assertions are represented using the assert keyword in alloy, this tells the alloy verification tool that every instance of the object must be shown to satisfy the constraint defined within the assert keyword

For example

```
        assert {
                amount > 0;
        }
```

Is translated into

```
	assert AccountAssert {
		all n: Account| (n.amount) > (0)
	}
```

Now note that this is not a solid way of verifying the program, because the way alloy work is to try to find instances where the assertion is false, it does not check how variables are changed after each function call.

Therefore, the ideal way of checking class invariants is to add the class invariant to the precondition and postcondition of every single function in the source file, and check if the postcondition of every function is valid, which we have not implemented yet.

Now here comes the challenging part, verifying the pre and post condition for function, the way we do it is to calculate the hoare triple for each function **(precondition => WP(functionBody, postCondition).**

Now for the weakest precondition we have implemented the base case (Assignment), the sequential combination case, and an if else statement case. You can see the detailed implementation in the **WPCalculator** class in the program package.

After we calculate the hoare triple, the way we verify that is to negate the hoare triple, and try to find an instance where the negation of the hoare triple is true. If we can find that, that means the postcondition is not true, otherwise we can say the postcondition might be true.

For example in **input-2.txt**

```
   func int getBalance() {
      y = balance;
      ensure {
         y == balance;
      }
   }
```

Have the weakest precondition (balance == balance), so the hoare triple is (true => (balance == balance)), and we negate that and try to find counterexamples. So this Is translated into:

```
pred getBalanceCheck{
	some n: Account| not ((n.balance) = (n.balance))
}
```

And once we run getBalanceCheck, we know if the postcondition for the function is valid.

Another small thing we need to handle is the boolean type for alloy, in alloy the **Bool** type is different from **PrimitiveBoolean**, so we use **(1=1)** to represent **true**, and **(1 = 0)** to represent **false**. We also use **(x = True)** to translate **Bool** type x into **PrimitiveBoolean** so you will notice

```
x <=> ((k == 3) || false)
```

Where x is a bool variable in the input language, will be translated into:

```
(x = True) <=> ((k = 3) or (1 = 0))
```

Which is logically equivalent to the original expression