Name	Name	Last commit date
max3_2.py	adds P4 student version	7 minutes ago
max3_3.py	adds P4 student version	7 minutes ago
mini_dafny.py	adds P4 student version	7 minutes ago
skip_0.py	adds P4 student version	7 minutes ago
skip_1.py	adds P4 student version	7 minutes ago
slow_copy_0.py	adds P4 student version	7 minutes ago
slow_copy_1.py	adds P4 student version	7 minutes ago
slow_copy_2.py	adds P4 student version	7 minutes ago
slow_square_0.py	adds P4 student version	7 minutes ago
slow_square_1.py	adds P4 student version	7 minutes ago
subst.py	adds P4 student version	7 minutes ago
swap_0.py	adds P4 student version	7 minutes ago
swap_1.py	adds P4 student version	7 minutes ago
test1.py	adds P4 student version	7 minutes ago
test2.py	adds P4 student version	7 minutes ago

README.md

Mini Dafny

In this project, you will implement a minimal version of a program verification tool, such as Dafny. You will implement the project in Python using Z3 solver APIs. The source program we want to verify is written in pseudocode. For example, the following code

```
assume (x + 3) * 2 >= 6
x := x + 3
x := x * 2
assert x >= 6
```

is written in pseudocode as

```
# declare variables
x = INT_VARS('x')

# code to verify
asgn = BLOCK(
  ASSUME((x+3) * 2) >= 6),
  x |◁·⩲|  x + 3,
  x |◁·⩲|  x * 2,
  ASSERT(x >= 6),
)
```

The following example code copies input to output by iterative addition:

```
var x,y,inp,out
assume (0 <= inp),
x := inp;
y := 0;
while 0 < x
invariant x >=0 && y + x  == inp
{
    x := x - 1,
    y := y + 1,
}
out := y,
assert (out == inp),
```

In psuedocode, it is written as:

```
x, y, inp, out = INT_VARS('x y inp out')
BLOCK(
  ASSUME(0 <= inp),
  x |◁·=| inp,
  y |◁·=| 0,
  WHILE(0 < x,
    # invariant
    AND(x >=0, y + x  == inp).smt_encode(),

    # body
    x |◁·=| x - 1,
    y |◁·=| y + 1,
  ),
  out |◁·=| y,
  ASSERT(out == inp),
)
```

Here is the list of commands in this pseudocode format.

Dafny command	pseudocode
assert	ASSERT
assume	ASSUME
if	IF
while	WHILE
{...}	BLOCK(...)
S1; S2	SEQ
empty command	SKIP
and	AND
or	OR

Also note that, `|◁·=|` denotes the assignment operator `:=`. We used the pseudocode and the specials symbols because we did add a parser for the source language.

The `verify` function in `mini_dafny.py` verifies a program. `verify` takes a code segment, an optional precondition for `requires` clauses, an optional postcondition for `ensures` clauses, and a flag to display verbose information.

```
def verify(stmt, pre = True, post = True, verbose = True):
  ...
```

Start with the example `skip_0.py` .

```
from mini_dafny import *

# We want to verify Hoare skip rule
#  assume x >= 0
#  skip()
#  assert x >= 0
#

# declare variables
x  = INT_VARS('x')

# code to verify
skip = BLOCK(
  ASSUME(x >= 0),
  SKIP(),
  ASSERT(x == 0),
)

# verify this code
verify(skip,verbose = True)
```

If you run

```
python3 skip_0.py
```

You should see the following result:

```
----------------------------------
Verification Condition
----------------------------------

Implies(True,
        Implies(0 <= x,
                And(Implies(0 == x, True),
                    Implies(Not(0 == x), False))))

----------------------------------
Simplified Verification Condition
----------------------------------

Or(0 == x, Not(0 <= x))

----------------------------------
Solver Response
----------------------------------
counterexample
[x = 1]
```

Z3 solver generated a counterexample because the given verification condition (formula) did not verify.

Verification Condition

We generate the verification conditions in backward direction. For the following code,

```
ASSUME(x >= 0),
SKIP(),
ASSERT(x >= 0),
```

we generate the following list of verification conditions.

Code	Formula
ASSERT(x >= 0)	And(Implies(0 == x, True), Implies(Not(0 == x), False))
SKIP()	
ASSUME(x >= 0)	0 <= x
Precondition True	True

From top to bottom, each row is implied by the previos row. Therefore, we have

```
vc = Implies(True,
        Implies(0 <= x,
                And(Implies(0 == x, True),
                    Implies(Not(0 == x), False))))
```

We can verify the verification condition with z3:

```
s = Solver()
s.add(Not(vc)) #formula is valid iff Not(F) is unsatisfiable.
r = s.check()
if r == unsat:
  print("proved!")
else:
  print("counterexample")
  print(s.model())
```

More Examples

Here is a list of examples that show the verification conditions.

- ASSUME

```
verify(ASSUME(x>0), True, True)

Implies(True, Implies(0 < x, True))
```

- SKIP

```
verify(SKIP(), True, True)

Implies(True, True)
```

- ASSERT

```
verify(ASSERT(x > 0), True, True)

Implies(True,
          And(Implies(0 < x, True),
             Implies(Not(0 < x), False)))
```

- BLOCK

```
verify(BLOCK(), True, True)

Implies(True, True)
```

- ASSIGN

```
a = INT_VARS('a')
verify(BLOCK(ASSIGN(a,a+1), ASSERT(a==1), ASSERT(a==2)),True,True)

Implies(True,
         And(Implies(1 == a + 1,
                     And(Implies(2 == a + 1, True),
                         Implies(Not(2 == a + 1), False))),
             Implies(Not(1 == a + 1), False)))
```

- WHILE

```
i,n = INT_VARS('i n')

# code to verify
code = BLOCK(
  i |◁·≒| 0,
  WHILE(i < n,
    # invariant
    AND(i >=0, i <= n).smt_encode(),

    # body
    i |◁·≒| i + 1,
  ),
  ASSERT(i == n),
)

# ask the solver to check this code
verify(code, verbose = True)
```

Verification condition

```
Implies(True,
        And(And(0 <= 0, 0 <= n),
            ForAll(i,
                   And(Implies(And(i < n,
                                   And(0 <= i, i <= n)),
                               And(0 <= i + 1, i + 1 <= n)),
                       Implies(And(Not(i < n),
                                   And(0 <= i, i <= n)),
                               And(Implies(i == n, True),
                                   Implies(Not(i == n),
                                        False)))))))
```

Simplified Verification Condition

```
And(0 <= n,
    ForAll(i,
           And(Or(Not(And(Not(n <= i), 0 <= i, i <= n)),
                  And(-1 <= i, i <= -1 + n)),
               Or(Not(And(n <= i, 0 <= i, i <= n)), i == n))))
```

Part 1

`mini_dafny.py` contains the code for generating the verification conditions and verifying it. Only part missing is the weakest precondition calculation. The function `wp(self, post)` takes a statement and a postcondition, and returns the precondition. Your job is to complete the function `wp(self, post)` in the following classes:

- BLOCK
- SEQ
- ASSIGN
- SKIP
- ASSUME
- IF
- ASSERT
- WHILE

Start with the simplest example `skip_0.py`. Currently, if you run `python3 skip_0.py`, it shows the program is proved. Actually the program fails to verify when `x = 1`. When you complete part 1, `python3 skip_0.py` will show

```
counterexample
[x = 1]
```

When you finish the part 1, run `python3 test1.py`. The tests in `test1.py` and `test2.py` should all pass. In a testcase, if the expected output is "unsat", that means this program should verify. The verifier could not find a counterexample. If the verifier finds a counterexample, it returns "sat". For example:

```
def test_slow_copy(self):
        import slow_copy_0
        actual = verify(slow_copy_0.code,verbose = False)
        self.assertEqual(actual, unsat)

        import slow_copy_1
        actual = verify(slow_copy_1.code,verbose = False)
        self.assertEqual(actual, sat)
```

`slow_copy_0` should verify and `slow_copy_1` should not verify.

Substitution in the Assignment

When claculating the weakest preconditon for an assignment, we replace the left-hand side variable in the assignment with the right-hand side expression of the post condition. Here is the Hoare assignment rule:

Hoare logic rule for assignment:

```
---------------------------- (hoare_asgn)
{ Q[x := a] }  x := a  { Q }
```

`{ Q[x := a] }` denotes the assertion Q in which each free occurrence of x has been replaced by the expression a. For example: to calculate the weakest precondition for the following Hoare triple, we replace the left-hand side x in the assignment with 1, right-hand side expressions of the post condition.

```
{ ??? }  x := x + y  { x == 1 }
{ 1 == x + y }  x := x + y  { x == 1 }
```

To implement this type of subsitituion, you can use the `substitute` function of Z3.

```
from z3 import *

x, y = Bools('x y')
expr = And(x, Or(y, And(x, y)))
print(expr)
#And(x, Or(y, And(x, y)))

expr2 = substitute(expr, (x, BoolVal(True)))
print(expr2)
#And(True, Or(y, And(True, y)))
print (simplify(expr2))
#y

a,b = Ints('a b')
expr3  = a * 2 + b * 3
print(expr3)
# a * 2 + b * 3
expr4 = substitute(expr3, (a, (b + 4)))
print(expr4)
# expr4 = (b + 4) * 2 + b *3
```

Part 2: (Optional) Create your own test case

Create a file called "student_tests.py" and add your own tests. If your test case breaks our canonical implementation, you will receive extra credit.

Submission

Upload the file `mini_dafny.py` and `student_test.py` (if you created your own tests) to gradescope.