HW1 Problem Set

ECS 261

Due: Friday, January 23, 2026

1. (a) Consider the following absolute value function written in Python:

```
def abs(x):
    if x >= 0:
        return x
    else:
        return -x
```

This problem will explore writing specifications for this simple function. Using Python functions and plain Python assertions, write executable specifications for the following properties. Each specification should have the following form, and assume that `x` and `y` are Python integers. The first spec is written for you.

```
def spec_abs_1(x):
    if x >= 0:
        assert abs(x) == x

def spec_abs_2(x, y):
    # your code here

def spec_abs_3(x, y):
    # your code here

def spec_abs_4(x):
    # your code here

def spec_abs_5(x, y):
    # your code here
```

Properties:

1. If x is greater than or equal to 0, then the absolute value of x is equal to x.
2. If x is less than y, then the absolute value of x is less than the absolute value of y.
3. If x is equal to $y+1$, then the absolute value of x is equal to 1 plus the absolute value of y.
4. The absolute value applied twice (absolute value of the absolute value of x) is equal to the absolute value of x.
5. The absolute value of $x+y$ is less than or equal to (the absolute value of x) + (the absolute value of y).

(b) Then, write tests for each of your five functions that test them on all integers `x` and `y` between −100 and 100. Each test should be named `test_abs_<n>` and have no arguments. That is,

```
def test_abs_1():
    # your code here

# etc.
```

Run your tests with `pytest hw1.py`. Which tests pass and which fail? If the property does not hold, add the pytest annotation to mark an expected failure:

```
@pytest.mark.xfail(reason="The property is not true")
```

(c) Come up with a spec and test for `abs` that is false, but happens to be true for integers between −100 and 100. Write your spec and test and verify that this is indeed the case. (Make sure your test is not marked as an "expected failure", as it won't fail!) Then write a `test_abs_6_fixed`, which increases the range of integers tested to fix the failure. Mark this one as an expected failure. Both tests should terminate within a few seconds.

```
def spec_abs_6(x):
    # your code here

def test_abs_6():
    # your code here

def test_abs_6_fixed():
    # your code here
```

Does this spec seem like a realistic case that might come up in practice? Why or why not?

(d) In a few sentences, comment on the effectiveness of this approach to testing. Is this more thorough than unit testing? What could go wrong?

2. This part will explore *stronger* and *weaker* specifications. Draw a graph containing your six specifications from Problem 1, together with the following three additional specifications:

- Spec 7: True
- Spec 8: False
- Spec 9: For $y = \texttt{abs}(x)$, either $y = x$ or $y = -x$; AND $y \geq 0$.

Your graph should contain 9 nodes, labeled 1 through 9, corresponding to the 9 specifications. Then draw an arrow from node i to node j if specification i is *stronger* than specification j. Include your graph in the output PDF.

Addendum: To make your graph more visually simple, you may omit an arrow $i \rightarrow k$ if you already have arrows $i \rightarrow j$ and $j \rightarrow k$.

3. It is possible to write specifications in Z3, rather than just in plain Python tests. For example, here is the absolute value function, this time written in Z3:

```
def abs(x):
    return z3.If(x >= 0, x, -x)
```

This function returns a Z3 formula. We can then write specifications for Z3 functions, by asserting that the resulting formula is provable. The `helper.py` file contains some helper functions to help this process, including the `PROVED`, `FAILED`, and `COUNTEREXAMPLE` constants. For example, specification 1 from Problem 1 can be asserted in Z3 as follows:

```
def test_abs_z3_1():
    x = z3.Int('x')
    spec = z3.Implies(x >= 0, abs(x) == x)
    assert prove(spec) == PROVED
```

If instead the specification was false, we would use:

```
    assert prove(spec) == COUNTEREXAMPLE
```

(a) Using this methodology, rewrite the following Python function which is used to update a player's level in a video game into Z3, then use Z3 to *prove* that the assertion in the Python function always holds: that is, the player level will always be between 1 and 100.

The idea is that once we have a proof, we can omit the assertion from production code, as it will always be true. You may assume as a precondition that the player level is previously between 1 and 100 when the function is called.

```
def update_player_level(player_level, delta):
    if delta < 0:
        result = player_level
    elif player_level + delta > 100:
        result = 100
    else:
        result = player_level + delta

    # This line is the assertion that we want to prove
    assert result >= 1 and result <= 100

    return result

def update_player_level_z3(player_level, delta):
    # TODO
    raise NotImplementedError

@pytest.mark.skip
def test_proving_assertion():
    # TODO
    raise NotImplementedError
```

(b) Based on this experience, do you think it would it be possible to automatically translate Python code to Z3 code to prove such assertions? Why or why not?

(c) What is one advantage and one drawback on the effectiveness of this approach to verifying specifications, compared to the testing approach in Problem 1?

4. A *source-to-source compiler* is a program that takes as input code written in a certain language, and produces as output code written in the same language. For example, we can consider source-to-source compilers that take code written in Python and produce output written in Python. Assume that any source-to-source compiler C should preserve the semantics of the Python code on any input in the following sense: any source program P on any input x should have the same observable behavior as the compiled program $C(P)$ on input x, and should produce the same output.

   Using the definition of specification that we saw in class, prove or disprove the following statements:

   (i) Given any source-to-source compiler C and any program P, and specification ϕ, if P satisfies ϕ, then $C(P)$ must also satisfy ϕ.

   (ii) Given any source-to-source compiler C and any program P, and any functional correctness specification ϕ, if P satisfies ϕ, then $C(P)$ must also satisfy ϕ.

   For the purposes of this question, assume that *observable behavior* includes I/O, any program state that is modified as a result of calling the function (e.g., mutation or global variables), system calls (filesystem or network I/O), and any other behavior that could be observed from calling the function as a black box, but not including running time or space usage.

5. The *four numbers game* works as follows: First, I secretly think of two positive integers x and y. I don't tell you what they are, but instead I give you four numbers: a, b, c, d and tell you that they are the values of the sum, difference, product, and quotient ($x + y$, $x - y$, xy, and x/y), in an unknown order. Assume that the difference is nonnegative and the quotient is a whole number. Can you figure out what x and y are?

   Examples:

   - Four numbers: $20, 95, 105, 500$. *Solution:* $x = 100$, $y = 5$.
   - Four numbers: $2, 6, 18, 72$. *Solution:* $x = 12, y = 6$.
   - Four numbers: $0, 1, 1, 2$. *Solution:* $x = 1, y = 1$.

   (a) Write a solver for this game using Z3. Your solver should be provided as two functions: the first function `q5a(a, b, c, d)` should, when given four Python integers a, b, c, d, return a solution as Python integers x, y, if there is at least one solution, or (`None`, `None`) if there is no solution. Second, the function `q5_run()` should provide an interactive version: it should prompt the user for 4 numbers, then display as output the correct answers `x` and `y`.

   We have provided a function `get_solution` in `helper.py` that will be useful for this part. If the spec is satisfiable (SAT), it will return a solution that you can use to get the values of x and y:

```
x = Int('x')
x_val = get_solution(spec)[x]
```

(b) Write a second solver that this time, determines whether the output is unique. `q5b(a, b, c, d, x, y)` should return a Python string "multiple", "unique", or "none" depending on whether multiple solutions exist - given the four Python integers and the output (x, y) from part (a) (x, y will be (`None`, `None` if there was no solution from part (a)). Update your `q5_run` interactive version to also show the output of `q5b`.

Submission instructions:

- Upload your solutions (as a PDF) and your code (in Python) in Gradescope.
- Your PDF should be a single file `hw1.pdf`. If you use this LaTeX template to create your solutions, please remove the problem statements and include only your solutions.
- Your code should be a file `hw1.py`, together with `helper.py` and any other necessary helper files. It should include your solutions for parts 1, 3, and 5.
- Please include all of the function names and signatures as listed in the document above, and do not modify any function signatures. You are welcome to add additional functions and tests – but if you do so, please include all additional tests only at the bottom of the file.
- If your code is correct, `pytest hw1.py` should run successfully without any errors (marked in red); instead, you should be using the annotation `pytest.mark.xfail` to mark tests that are expected to fail. These will show up in yellow.