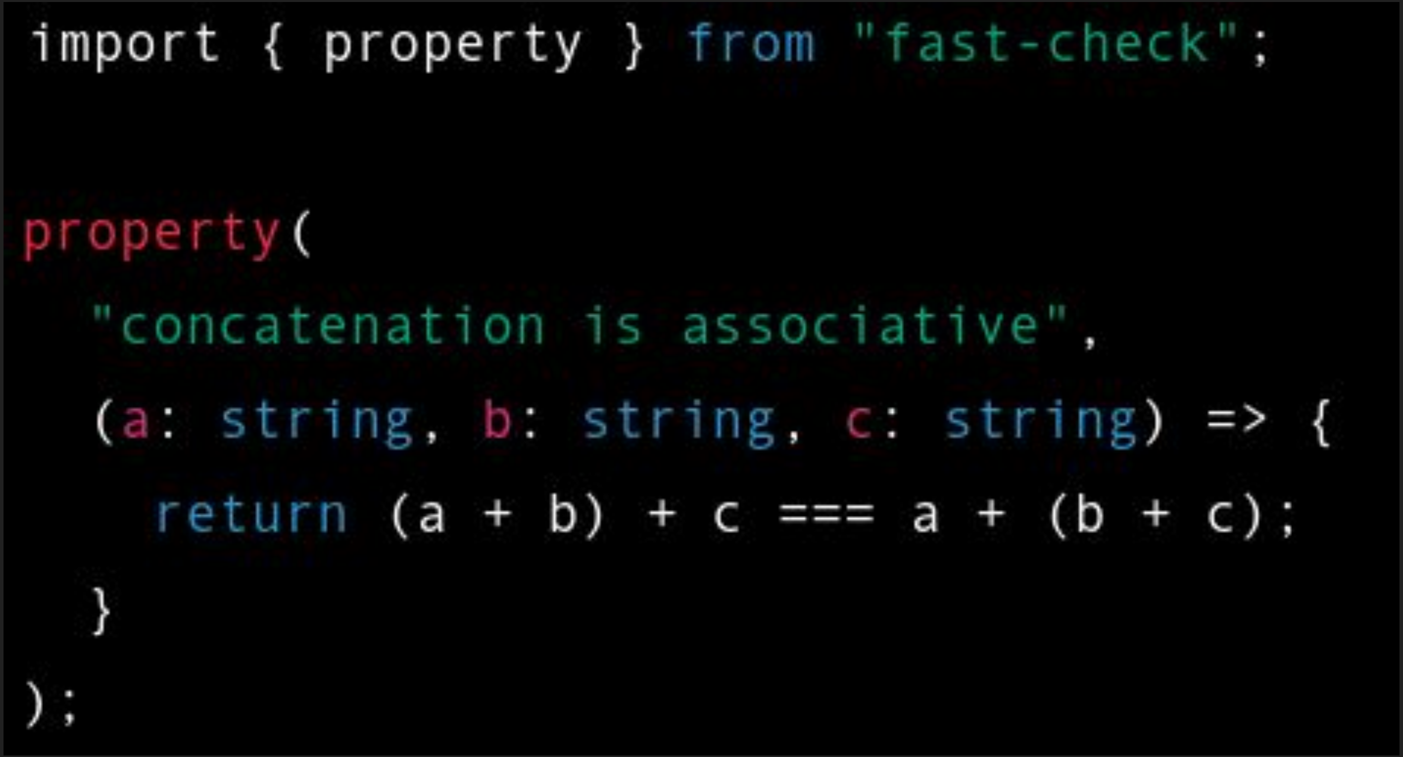
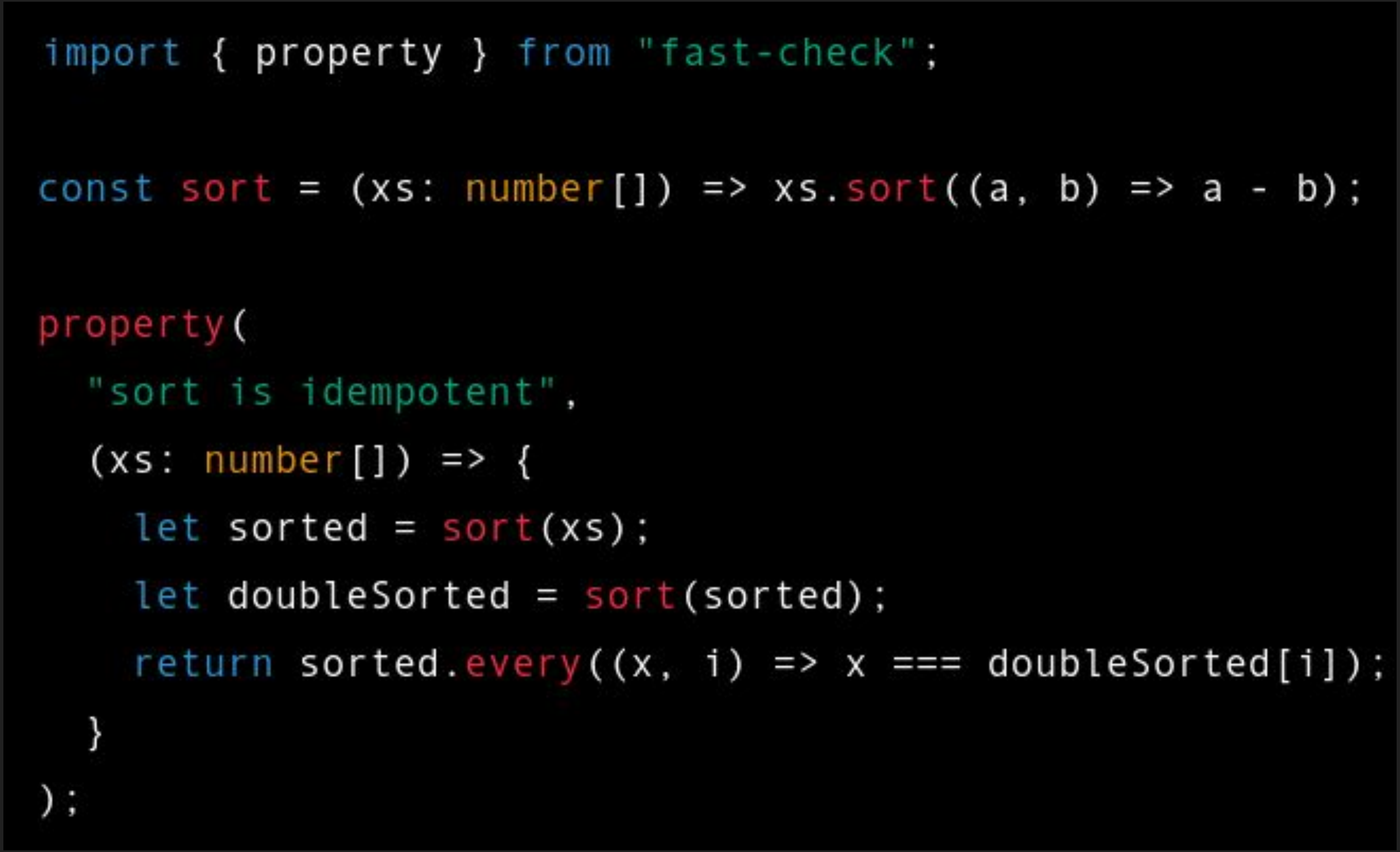
Properly 🅱️ased testing

For joke reasons, this is legal

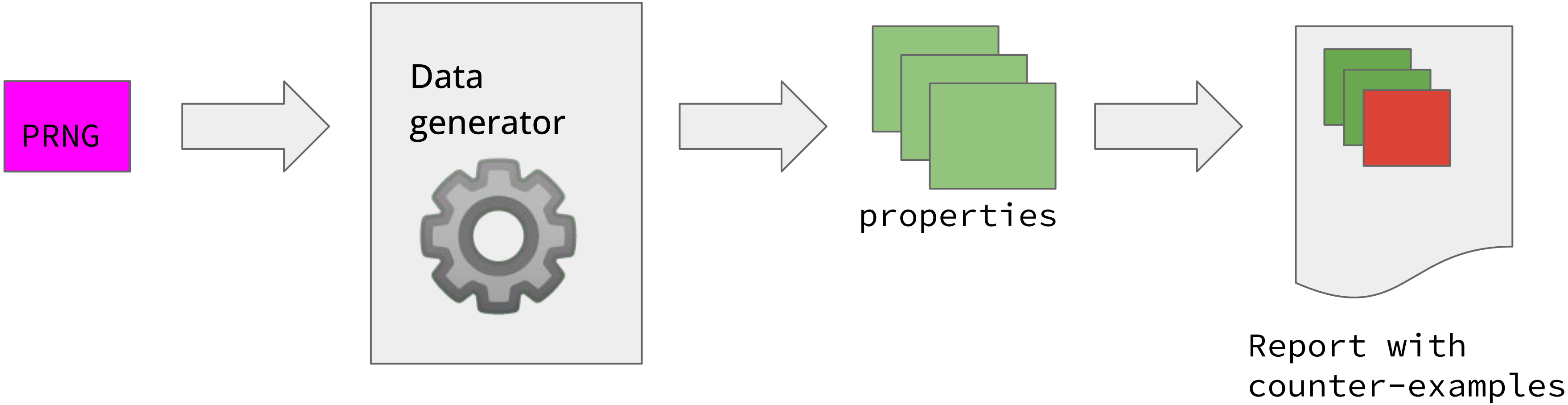
SOME TYPESCRIPT THING



```
import { property } from "fast-check";


property(
  "concatenation is associative",
  (a: string, b: string, c: string) => {
    return (a + b) + c === a + (b + c);
  }
);
```



```
import { property } from "fast-check";

const sort = (xs: number[]) => xs.sort((a, b) => a - b);

property(
  "sort is idempotent",
  (xs: number[]) => {
    let sorted = sort(xs);
    let doubleSorted = sort(sorted);
    return sorted.every((x, i) => x === doubleSorted[i]);
  }
);
```



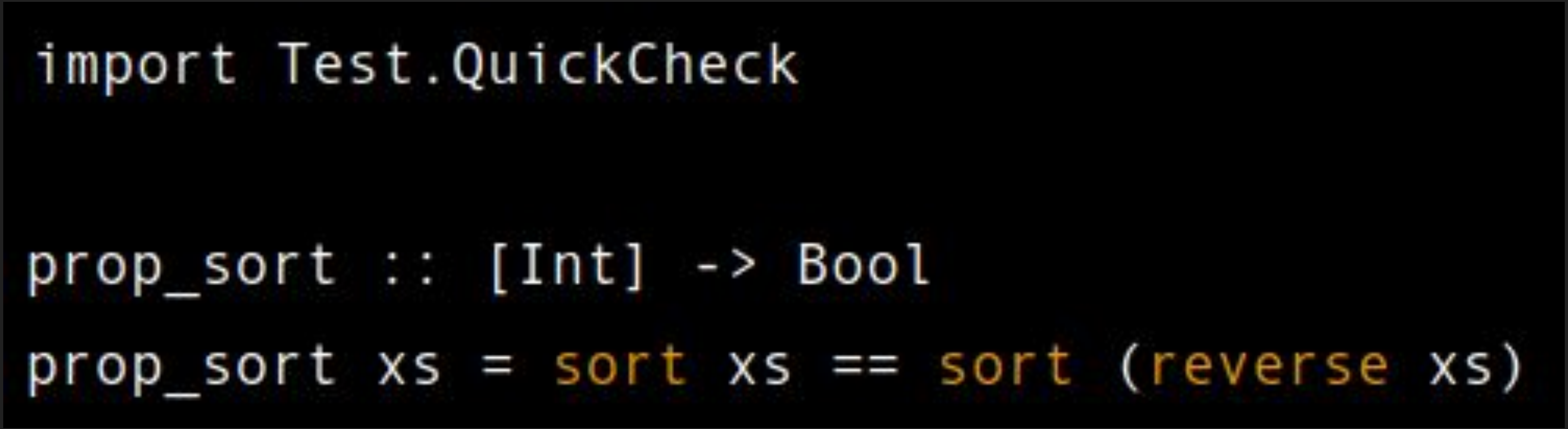
PRNG

Data generator

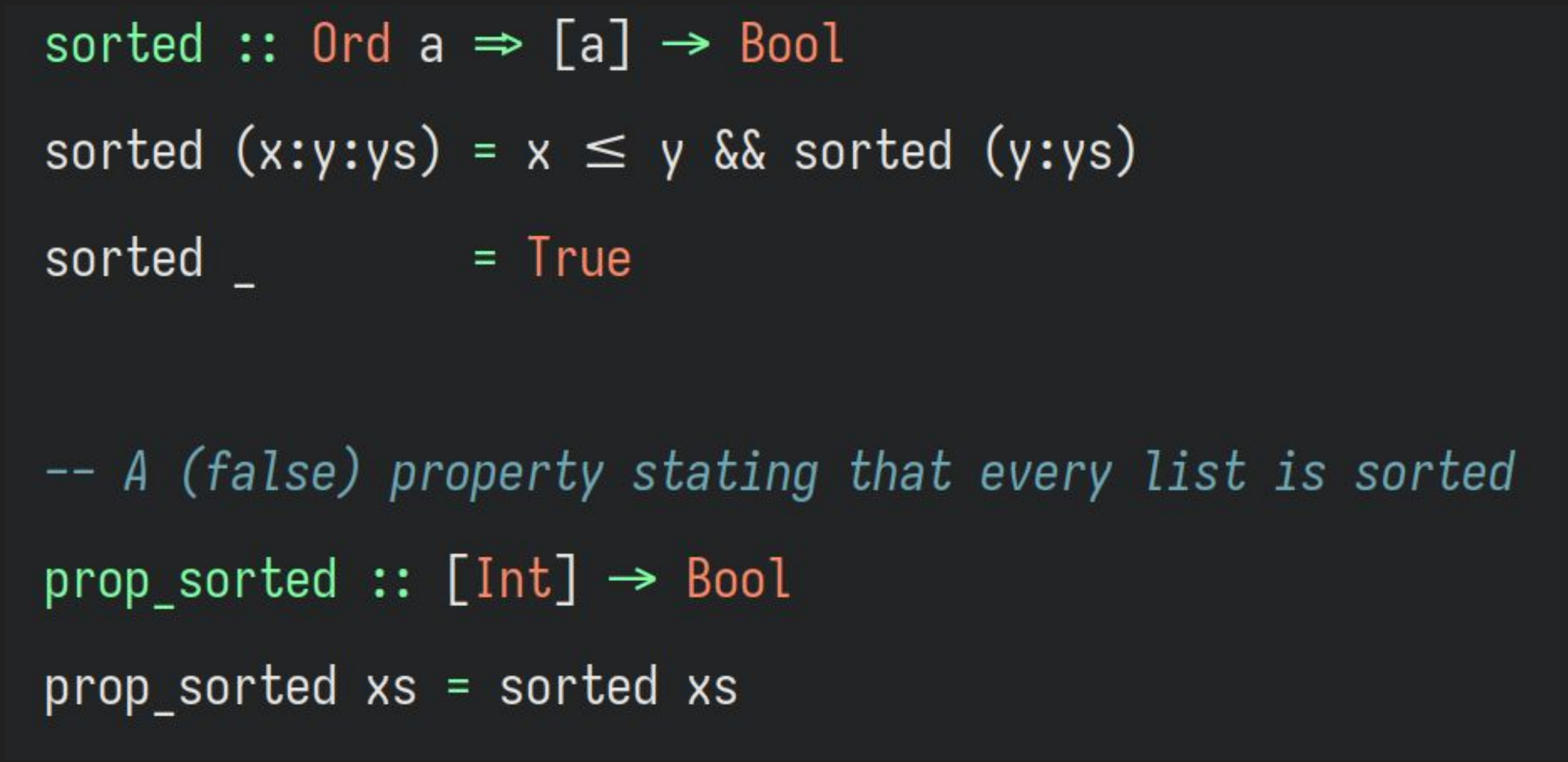
properties

Report with counter-examples

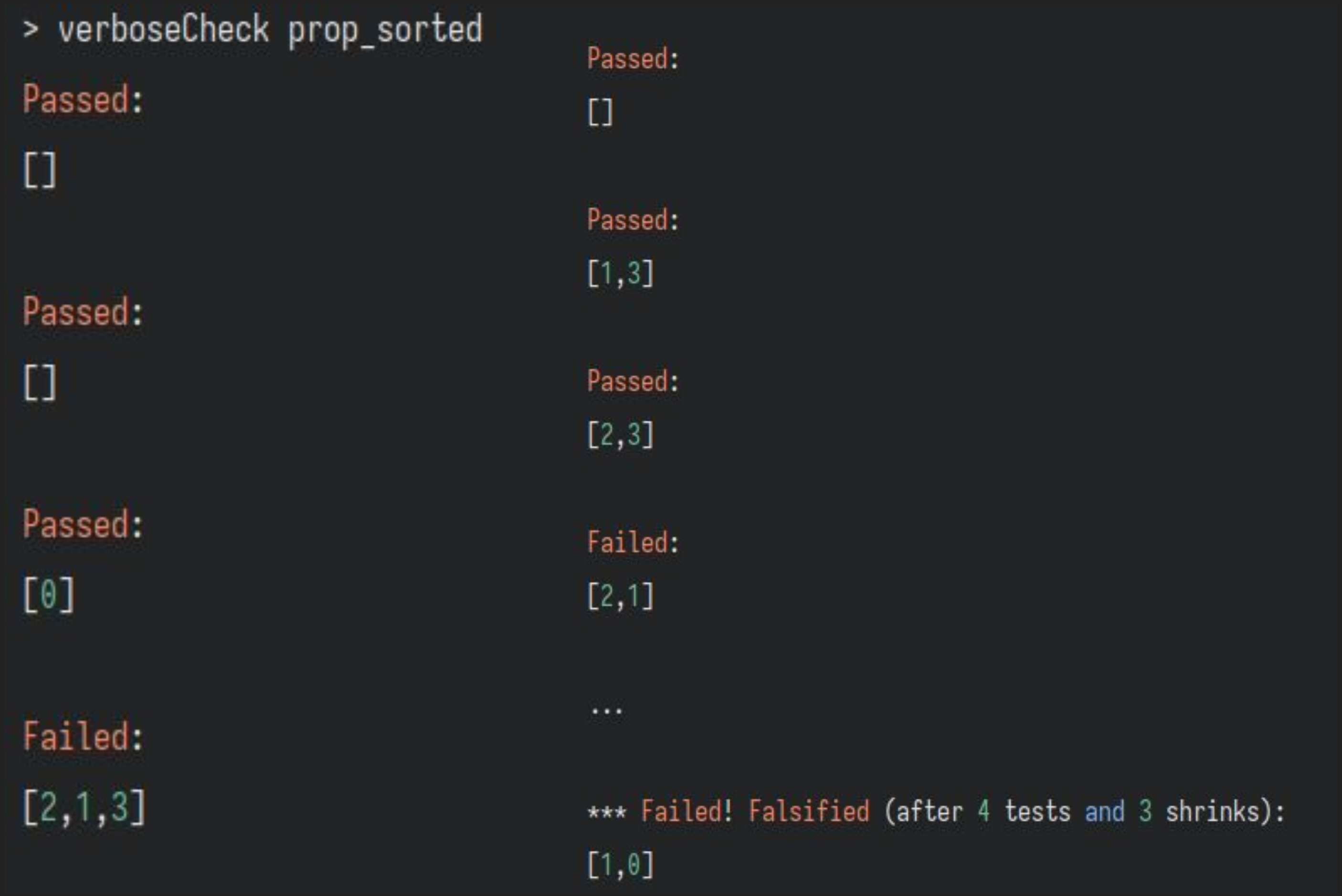
QuickCheck



```
import Test.QuickCheck


prop_sort :: [Int] -> Bool
prop_sort xs = sort xs == sort (reverse xs)
```

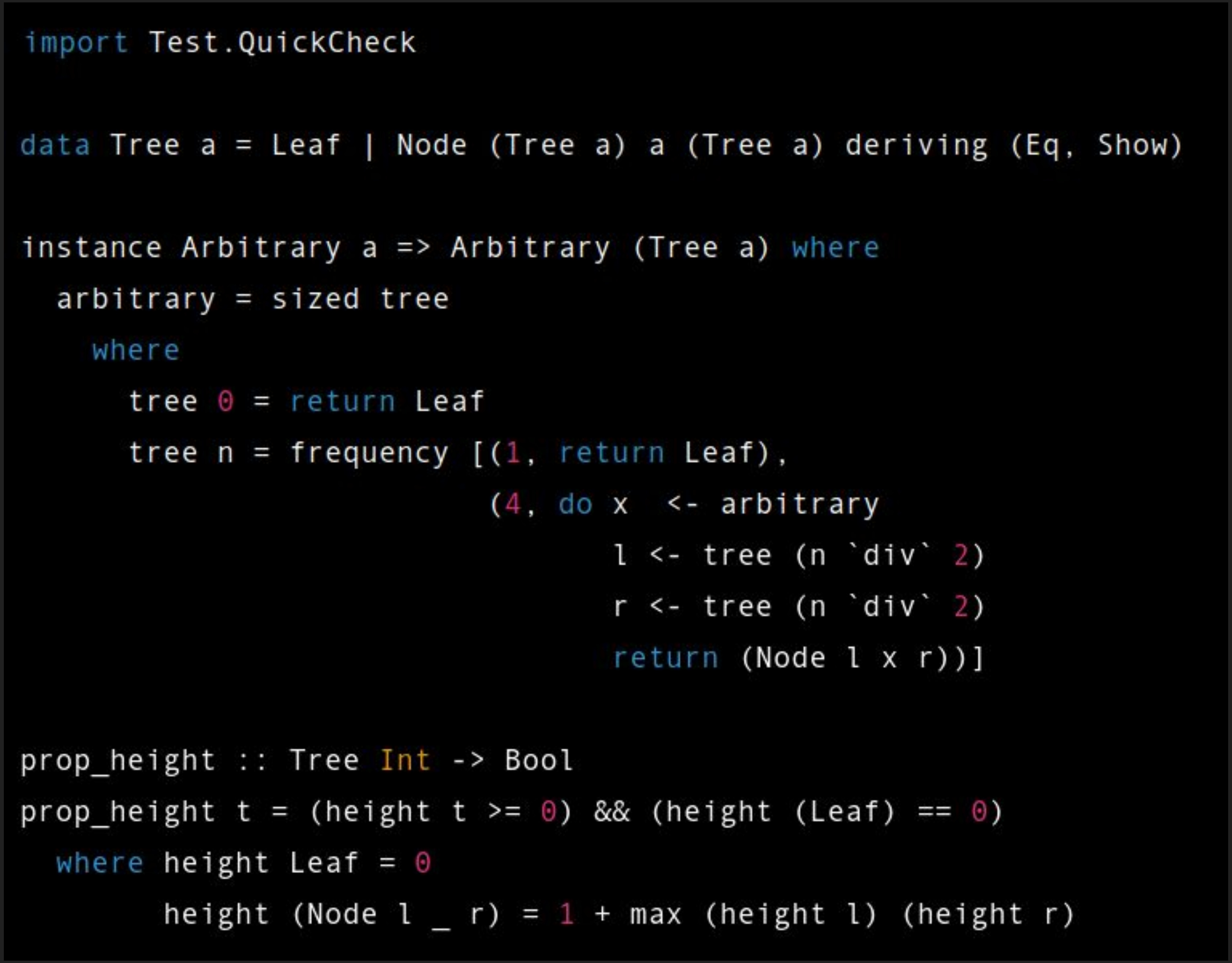


```haskell
sorted :: Ord a ⇒ [a] → Bool
sorted (x:y:ys) = x ≤ y && sorted (y:ys)
sorted _        = True


-- A (false) property stating that every list is sorted
prop_sorted :: [Int] → Bool
prop_sorted xs = sorted xs
```



```
> verboseCheck prop_sorted
Passed:
[]

Passed:
[]

Passed:
[0]

Failed:
[2,1,3]
```

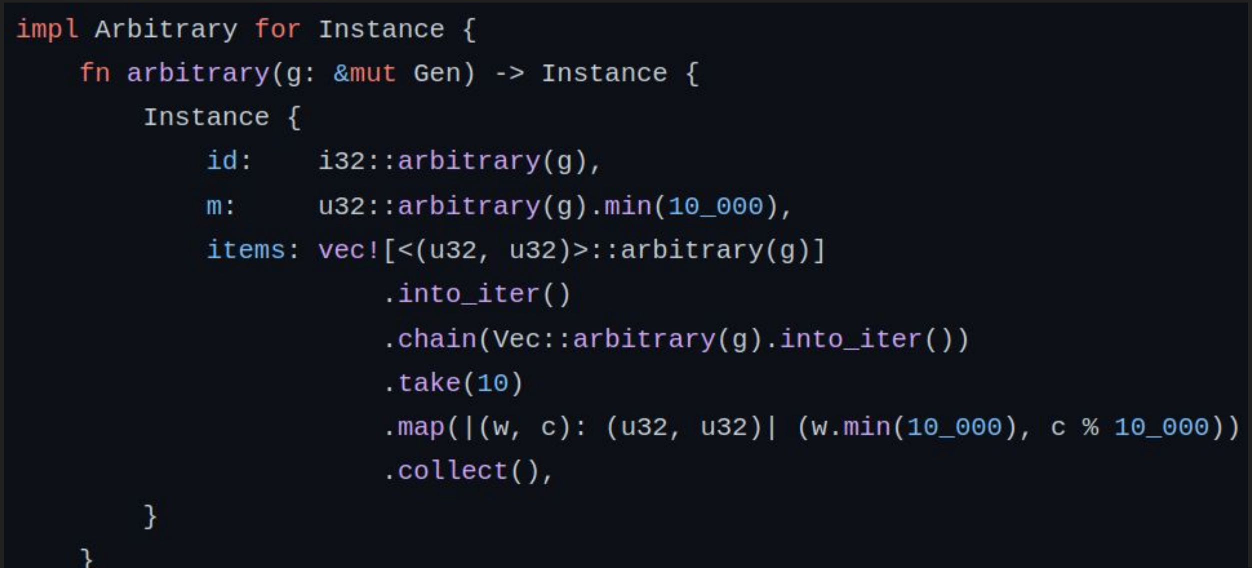
```
Passed:
[]

Passed:
[1,3]

Passed:
[2,3]

Failed:
[2,1]

...

*** Failed! Falsified (after 4 tests and 3 shrinks):
[1,0]
```



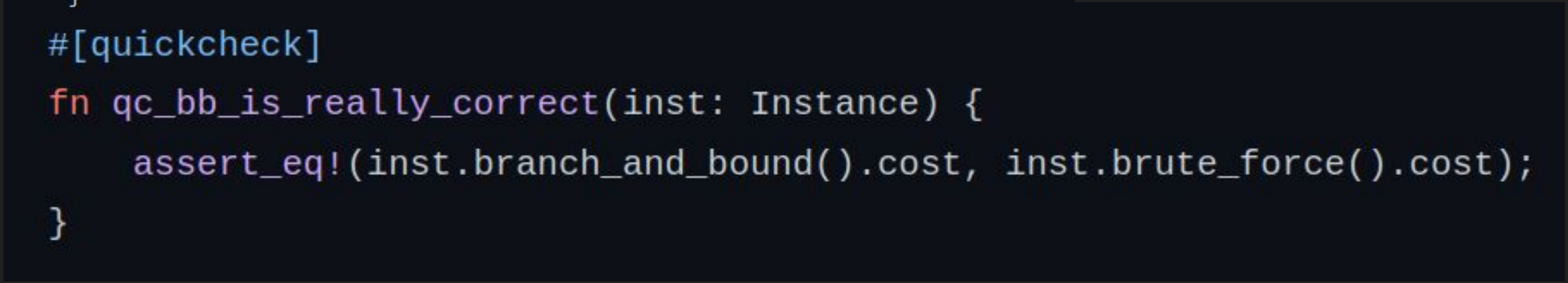
```haskell
import Test.QuickCheck

data Tree a = Leaf | Node (Tree a) a (Tree a) deriving (Eq, Show)

instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized tree
    where
      tree 0 = return Leaf
      tree n = frequency [(1, return Leaf),
                          (4, do x  <- arbitrary
                                 l <- tree (n `div` 2)
                                 r <- tree (n `div` 2)
                                 return (Node l x r))]

prop_height :: Tree Int -> Bool
prop_height t = (height t >= 0) && (height (Leaf) == 0)
  where height Leaf = 0
        height (Node l _ r) = 1 + max (height l) (height r)
```

MOAR CODE!

https://tinyurl.com/properly-based-kop



```
impl Arbitrary for Instance {
    fn arbitrary(g: &mut Gen) -> Instance {
        Instance {
            id:    i32::arbitrary(g),
            m:     u32::arbitrary(g).min(10_000),
            items: vec![<(u32, u32)>::arbitrary(g)]
                        .into_iter()
                        .chain(Vec::arbitrary(g).into_iter())
                        .take(10)
                        .map(|(w, c): (u32, u32)| (w.min(10_000), c % 10_000))
                        .collect(),
        }
    }
```



```
#[quickcheck]
fn qc_bb_is_really_correct(inst: Instance) {
    assert_eq!(inst.branch_and_bound().cost, inst.brute_force().cost);
}
```

Common techniques

- Arbitrary data generators
- Smart arbitrary instances (corner-cases first)
- Multiple strategies
- Shrinking algorithms (state space explosion)
- Test case limits
- Failure thresholds
- Performance invariants
- Models of concurrency
- Persistent PRNG state
- Combine with unit testing

Drawbacks

- Generating structured data is hard (ASTs)
- Bugs in the test suite (invalid data)
- Limited generators may give false confidence
- A test is not a proof!

Use property-based testing!

- It’s often surprisingly easy to provide generators
- Shrinking rules
- No need to solve the general problem
- It works even in c++

Random advice

- Value in normal form must never capture variables