Logic for Programmers (free sample)

(version 0.11)

Hillel Wayne

Aug 04, 2025

Acknowledgements

Thanks to Tsvetan Tsvetanov, Predrag Gruevsky, Jeremy Kun, Saul Pwanson, Yeray Cabello, Igor Kuvychko, Puikei Cheng, Daniel Prager, Sviatoslav Abakumov, Harald M Müller, and Marianne Belloti for feedback on content. Thanks to David Mazarro, Jeroen Heijmans, Nirmalya Sengupta, Mike Mull, and Marcus Millin for identifying errors in the text.

Thanks to Tikhon Jelvis for some advice on covering constraint solvers.

Thanks to Zac Hatfield Dodds for helping with Sphinx and Alexey Zubkov for advice on typesetting.

Tim Nelson, Marianne Belloti, and Saul Pwanson all helped me keep on schedule at various points in this book's production.

Contents

Chapter 1

Free Sample

This free sample includes the *introduction* (page 2) and the chapter on *testing techniques* (page 5).

Chapter 2
Intro

2.1 Beta Notes

I'm doing early access with this book, so this is all beta. Most of the material is now in, but I still need to polish and revise it, add more exercises, improve formatting, and incorporate reader feedback.

I welcome any and all comments. I'm particularly interested in:

1. Do the examples seem useful to you? Were the exercises helpful?
2. Which topics need the most focus?
3. What resources would be good to recommend as "further reading"?
4. What examples and new topics would you like to see?
5. What needs more exercises?

You can email me at h@hillelwayne.com. Thank you very much!

Note

Anything in a note box is a message from me to you as early readers. Things I'm uncertain about, things I plan to polish more, things that I plan to write, etc. [[double braces]] are similar. Feel free to throw comments my way!

2.2 New in v0.11:

- Brand new chapter, "Proving Code Correct", covering proofs, loop invariants, formal verification
- Total rewrite of "Database" chapter:
  - Now covers database representations, relational model, queries, joins, and constraints
  - Two new executable SQL examples on constraints
  - One new image
- Total rewrite of "Functional Correctness":
  - Now covers assertions, MISU, polymorphism, advice

  - Loop invariants and formal verification moved to proofs chapter
- Total rewrite of "Case Coverage", now called "Case Analysis":
  - New introduction and motivating example
  - More material on analysing code with decision tables, techniques, when not to use DTs
  - Redundant examples removed
- Logic chapter improved, now covers the way-more-common scoped quantifiers before unscoped
- Fixed "symmetric difference" exercise
- Six exercises removed, eleven added (+5 total)
- Better format for proof tables and rewrite rules
- Some initial table of contents tweaks
- Fixed PDF bug: admonition sidebars now render correctly in Acrobat

2.3 Why this book

> If I start a build at 3:05 PM and it takes 12 minutes to complete, when will the build be finished?

To answer this question, we need to how to manipulate numbers. The mathematics of numbers is called *arithmetic*. Arithmetic shows us how to multiply two numbers, use fractions, determine which of two numbers is larger, and more.

> If I have the conditional `if(sensor_offline || inactive)`, and I know for sure that `sensor_offline` is true, does the value of `inactive` matter?

To answer this question, we need to know how to manipulate booleans. The mathematics of booleans is called *logic*. Logic shows us how to simplify a boolean expression, use sets, determine if one statement is stronger than another, and more.

But there is one key difference between arithmetic and logic. We were taught arithmetic in elementary school. Few of us were formally taught logic. Most programmers pick up a little logic by osmosis, but even that rarely exposes people to anything beyond the basics.

This makes logic the single most useful topic in math a programmer can learn. But how are we supposed to learn it? There are plenty of books available written for philosophers, mathematicians, and computer scientists, who all have far more need for the theory than the practice. There are no books on logic meant for the self-studying programmer, who is looking for practical skills useful in day-to-day work. It is as if nobody will teach us how to ride a bicycle, only how to build one.

That is the goal of this book. I aim to teach you the basics of logic and how to apply it to various everyday software problems, like testing code, designing a database, or working out customer requirements. By the end of this book, you will be comfortable manipulating logical expressions and have a greater understanding of all of the ways software uses logic, implicitly or not.

2.4 Design Philosophy

This book is meant specifically for programmers with little to no familiary with formal math. In all cases, I opted for accessibility and ease-of-use over precision or rigor. This is a technical how-to, not a textbook.

2.4.1 Notation

Mathematics shares many operations in common with programming but uses different representations, such as writing "and" as $\wedge$ instead of &&. I will using programming terminology wherever possible. I included an appendix which maps conventional programming symbols to math symbols.

In cases where math symbols don't have common programming analogs (such as $\forall$), I have opted to use an explicit English equivalent (such as all).

Lastly comes the question of array indexing. Does the array arr start at arr[1] or arr[0]? There is no universal programming convention, as different languages make different choices. I would use the mathematician's convention except that does not exist either: different branches of mathematics make different choices too! So I will default to 0-based indexing unless demonstrating a tool or language which uses 1-based indexing, which I will explicitly note.

2.5 How to Read This Book

I recommend first reading chapter_predicates, and then moving to whichever technique looks most interesting. Techniques chapters are independent except when otherwise noted, in which case backreferences are provided.

The first five techniques (starting with chapter_conditionals) focus on how logic applies to everyday software. The last four (starting with chapter_data_modeling) cover special logic-based tools that unlock powerful new solutions to difficult software problems.

Large code samples are available online at https://github.com/logicforprogrammers/book-assets.

2.5.1 Exercises

Exercises are provided to help you check your knowledge and develop your skills further. All exercises have solutions in the back of the book. Some of the exercises have multiple possible solutions. Your answer can be correct even if it differs from the “official” solution!

Some questions involve writing short snippets of code. In these cases, use whatever language you like. I will personally give examples in Python or pseudocode. When writing Python, I have tried to make it as accessible as possible, meaning it does not do things in an idiomatic way.

Chapter 3

Writing Better Tests

The most common form of software test is the "example" test: pass an input into a function and check that it returns the correct output. Here are some example tests for max:

```
test max([1, 2, 3]) == 3
test max([1, 3, 2]) == 3
test max([2, 3, 3, 2]) == 3
```

Example tests are easy to write, but they are also limited. Many functions that are *not* max pass these tests:

1. A function that returns the largest absolute value in the list
2. A function that returns the most common element, breaking ties with max value
3. A function that returns the maximum of the first five elements
4. A function that just returns 3.

The more examples we write, the more invalid functions we rule out. But this is tedious and error prone. Logic provides us an alternative: express the essential meaning of the function, and then use this to generate hundreds of tests for us.

3.1 Strong and Weak Tests

Some tests are stronger than others.

"Stronger" has a precise logical meaning. This is because *tests are equivalent to predicates.* The first test in the last section is equivalent the same as the predicate P = max(l) == 3. The test passes when P is true and it fails when P is false. For convenience, I will use "P" to refer both to the predicate and the corresponding test.

This means we can use our same logical operators to express statements about tests. P && Q is true if (the tests corresponding to) P and Q both pass. P || Q is true if at least one of the two test passes. !P is true if the test P fails. Finally, P => Q (aka !P || Q) is true if P passing implies that Q also passes.

What does that mean in practice? It means that there is no possible version of max that *passes* P and *fails* Q. If a failing test means a buggy implementation, then any bug that "slips past" P will slip past Q, too. This means that P is at least as *strong* as Q, which is totally captured in the logical expression P => Q. If P can catch a bug that Q will miss, then P is *stronger*. As an example:

```
P = max([1, 2, 3]) == 3

Q = max([1, 2, 3]) >= 0

R =
  1. max([1, 2, 3])  >= 0
  2. max([0, 1, -1]) >= 0
```

If P passes, Q also passes. If R passes, Q also passes. This means P => Q && R => Q. We can further see that both are stronger than Q. Notice that P and R are stronger than Q in different ways: P gives a more specific answer for the same input, while R tests a wider range of inputs. Finally, neither R nor P are stronger than each other: each will pass some version of max the other would reject. Mathematicians would say that => forms a "partial ordering".

[[TODO graphical diagram of this]]

Exercise 1 (Partial Ordering)

1. Give a buggy implementation of max that R passes but P fails, and a buggy implementation that P passes and R fails.
2. Modify the two clauses of R to create a test T that's stronger than both P and R. It should fail both implementations you wrote above.
3. How would you express "T is at least as strong as both P and R?" Does this mean T is stronger than Q, too?

Solution (page 16)

Exercise 2 (The Flaw with False)

For any predicate P, false => P. So any possible bug in max that's caught by a test will also be caught by test false, making it the strongest possible test imaginable. And in fact Explain the flaw in this reasoning.

Solution (page **??**)

While P and R are stronger than Q, neither is strong enough, on its own, to guarantee that max is correct. This version of max passes both tests but still is incorrect:

```
max(list) =
  if list == [1,2,3] then 3 else
  if list == [0,1,-1] then 1 else
  -infinity
```

We now have two separate ways of making a test stronger: widen the number of inputs it tries, or make more specific claims about the outputs. The most powerful possible test would try every possible valid input to max and make the most specific claim possible about the output. We would call such a test a *total specification* (or total "spec") of max. It would pass if *and only if* max was correctly implemented, making any other kind of direct testing redundant. In other words, if T is *any* test of max, then TotalSpec => T.

What then, would be that test?

3.1.1 Specifying a function

First we need to define the *domain* of max- the set of all valid inputs. For the purposes of this chapter, I'll say max should work for any nonempty, noninfinite list of integers. The total specification looks like this:

```
IsMax(x, l) = `x is the maximum value in l`.

TotalSpec =
  all l in NonEmptyIntegerLists:
    IsMax(max(l), l)
```

It sometimes convenient for our purposes to restrict the domain to something expressable in a language's type system.

```
all l in IntegerLists:
  len(l) > 0 => IsMax(max(l), l)
```

Now we have to define what it means to be the "maximum value" of a list. First of all, it has to be an element of the list: if we take the max value and add ten, we no longer have the max value. Second, no element of the list is larger than it. It is easier to see how to formalize this property if we start by defining IsMax for *sets*:

```
IsMax(x, set) =
  1. x in set
  2. `no element of the set is larger that x`
```

Another way to say "no element is larger than x" is to say "for all elements y in the set, x is as least as big as y." That looks like an all to me!

```
IsMax(x, set) =
  1. x in set
  2. all y in set:
       x >= y
```

[[Programmers work in lists, not sets. We can't use quantifiers on lists, but we can instead use them on the set of their indices]]:

```
IsMax(x, list) =
  some i in 0..<len(list):
    1. list[i] = x
    2. all j in 0..<len(list):
         x >= list[j]

TotalSpec =
  all l in NonEmptyIntegerLists:
    IsMax(max(l), l)
```

Try writing a few valid tests for max, and then see if they are implied by TotalSpec.

Exercise 3 (Uniqueness)

Write the predicate IsUnique(l), which is true iff every element of l is unique. IE

```
IsUnique([1, 2, 3])
!IsUnique([1, 2, 1, 3])
```

Solution (page 16)

3.2 In Practice: Property-Based Testing

Reminder that our total specification for max was this:

```
IsMax(x, list) =
  some i in 0..<len(list):
    1. list[i] = x
    2. all j in 0..<len(list):
         x >= list[j]

TotalSpec =
  all l in NonEmptyIntegerLists:
    IsMax(max(l), l)
```

Implementing IsMax in our favorite programming language is straightforward, as is calling max on a list and checking that the output passes IsMax. Trying this for *all* infinity non-empty integer lists is impossible (at least without some tools covered in the next chapter). What we could do as a substitute is test one hundred randomly generated different lists. This would not be as strong as TotalSpec, but it would be much stronger than max([1,2,3]) == 3.

This is the idea behind *Property-Based Testing* (PBT). We first write a test that applies to any possible input, and then we randomly generate inputs to test it. There are some engineering details to figure out ("how *do* we generate non-empty integer lists?"), but most languages have a high level libraries that handle these details for us. Here's an example, using the python library Hypothesis[1]:

```
import hypothesis.strategies as s
from hypothesis import given
@given(s.lists(s.integers(), min_size=1))
def test_max(l):
    max_val = f(l) # our max function
    assert max_val in l                    # (a)
    assert all(max_val >= x for x in l)    # (b)
```

The @given is a generator ("strategy" in hypothesis' terms) that says the input can be any nonempty list of integers. We define all of the function's inputs this way, pass them to the test, run the function normally, and get the output. Finally, we check if the output satisfies our specification.

Tip

Reminder, you can download this code sample directly from https://github.com/logicforprogrammers/book-assets.

Compare that to our total specification. The quantified set NonEmptyIntegerLists becomes the generator (only test nonempty lists) and the body of the quantifier becomes our assertions.

In addition to handling the random generation, Hypothesis also gives us some convinces. In addition to purely random lists, it will also try common pathological cases. If an input fails, it will "shrink" the failing input to a smaller, simpler failing input. For example, if my implementation of max looked only at the first five elements of the list, here's what it could give me back:

```
Falsifying example: test_max(
    l=[0, 0, 0, 0, 0, 1],
)
```

[1] https://hypothesis.works/

Finally, Hypothesis stores a database of known failures and retries them on future runs.

Exercise 4 (Property Testing Find)

Look into whatever your favorite language's PBT library is, and then write a property test for find. You may have to write your own version myfind for your language, if the builtin does something besides return -1 for a missing value (like raise an exception).

Solution (page 17)

3.3 Notes on Property Testing

3.3.1 Partial Specifications

A *partial specification* is any spec that is covered by a total spec, ie any test where TotalSpec => PartialSpec. Every test we have seen so far *besides* test_max is a partial specification.

In theory, we should never need to test a partial specification. In practice, the majority of the tests we write are partial for two reasons. [[One, most of the functions we work with in software are too complex to be easily total specifiable.]] And even if we can totally spec a function, partial specs help us localize the source of bugs. A total spec failing tells us that the function is incorrect, but a partial spec failing tells us *why* it's incorrect.

For this reason, using property testing well means coming up with strong, testable partial specifications. Most functions will have at least *something* expressible, often to do with the domain of the problem:

- A dating app's match function shouldn't match people with cats to people with cat allergies.
- Making a chess move and undoing it should return us to the original game state.
- A customer who clicks "submit payment" ten times should only be charged once.
- If we cut frames 126-143 of a video, the output will be seventeen frames shorter and the 906th frame will now be the 889th.

Note

I could probably make those exercises.

There are also universal "tactics" that apply to many different problems in many different domains. One of the simplest and most famous tactics: the code does not crash on some input. This is called *fuzzing* and is very popular for low-level code (where memory leaks can lead to security vulnerabilities) and parsers. Similarly, we could test that no queries made to an API return a 500 error. If we have exception handling in code, we can test that only "expected" inputs raise errors, and that no other errors are raised.

Refactoring with Tests

Another popular tactic is "our function matches a reference function".

```
all x: f(x) = g(x)
```

Why might I want to test that I have two identical functions? One common reason is that I might have a simple function that solves my problem, but is too slow for production. I can use that to test a faster-but-more-complex version of the same function. Or I might have a simplified function that works for the happy path, and I want to make sure an edge-case-handling version still gets the same results on "good" inputs.

My favorite use-case, though, is testing that a refactoring did not change the code's behavior. We can take an example from the last chapter and show exactly that.

```
import hypothesis.strategies as s
from hypothesis import given

def old_function(l, P, Q):
  if not all(P(x) for x in l) or any(not Q(x) for x in l):
    return 1
  else:
    return 2

def refactor(l, P, Q):
  if all(P(x) and Q(x) for x in l):
    return 2
  else:
    return 1

@given(s.lists(s.integers()),
```

(continues on next page)

(continued from previous page)

```
        s.functions(like=lambda x: ...,
                    returns=s.booleans(), pure=True),
        s.functions(like=lambda y: ...,
                    returns=s.booleans(), pure=True)
        )

def test_max(l, P, Q):
    assert old_function(l, P, Q) == refactor(l, P, Q)
```

Notice that Hypothesis is able to randomly generate *functions*. These behave somewhat like mocks or stubs in unit testing: they are set to take any number of parameters and return a boolean value. In one run P might return False for every integer, in another it might return True for integers -1, 15, and 7.

Running this test shows that for all values, the simplified version of our function returns the same result.

Other tactics

One of the most famous tactics is the "round-trip" property, that converting data into another format and then back doesn't change the data.

```
Roundtrip(x_to_y(x), y_to_x(y)) =
  all x in X: y_to_x(x_to_y(x)) = x
```

The polars dataframe library found a bug this way. They generated dataframes, converted the columns into python lists, then converted the lists back into dataframe columns. The roundtrip is that column -> list -> column should give back the original column. Hypothesis found that this could drop timezones.

Roundtrip properties are generally effective when you have a custom datatype you want to convert into a portable format, like json or CSV.

A final useful class of tactics is "metamorphic properties". These are properties that relate multiple function calls together. For example, if your computer vision system recognizes an object, it should recognize the same object if you tilt the picture by two degrees. Or if you have query API with filters, adding a new clause to a filter should give you a subset of the results you get without it (a real bug this found in Spotify).

3.4 Summary

1. A function *specification* (page 6) is a mathematical description of how it behaves and its properties. Specifications can be full or partial.
2. Specifications can determine what are valid inputs and how they relate to outputs.
3. Tests ultimately check that a function matches its specification. Unit tests do this by checking a single input. Property tests instead generate lots of random inputs and check they all satisfy the properties.
4. We may not be able to get a full specification for your function, but we can still usefully use partial specifications.
5. Not all properties are function-local. Some span multiple functions or inputs.

As useful as PBT is, the idea that functions have specifications goes further. With it, we can expand on the idea of using specifications to verify the correctness of larger sets of code.

3.4.1 Learn More

[[Talk about fuzzing, quickcheck here, model-based testing]]

- Property Testing with Complex Inputs: https://www.hillelwayne.com/post/property-testing-complex-inputs/
- In Praise of Property Testing: https://increment.com/testing/in-praise-of-property-based-testing/
- The Fuzzing Book: https://www.fuzzingbook.org/html/Fuzzer.html
- Choosing properties in practice: https://fsharpforfunandprofit.com/posts/property-based-testing-3/
- Metamorphic Testing: https://www.hillelwayne.com/post/metamorphic-testing/

Chapter 4

Answers to Exercises

Answer to Exercise 1 Partial Ordering

Recall that P tests that the max of [1,2,3] is 3, while R tests that max values of [1, 2, 3] and [0, 1, -1] are >= 0.

1. To pass R and not P, write a max(l) = 1 . To pass P and not R, write a max that just returns the last value of the input. 2.

```
T =
  1. max([1, 2, 3]) == 3
  2. max([0, 1, -1]) == 1
```

   This fails both buggy max implementations given above.

3. "T is as strong as P and R" is T => P && R. Since P => Q, T => Q too, meaning T is as strong as Q.

Answer to Exercise ?? The Flaw with False

test false will reject any buggy implementation of max… but it will also reject a correct implementation! What makes a given test "a test of max" is that it will pass for a correct implementation, meaning false isn't a test of max at all (and cannot be stronger than them).

By contrast, test true *is* a valid test of max, and in fact the weakest possible test.

Answer to Exercise 4 Uniqueness

```
IsUnique(l):
  all x, y in 0..<len(l):
    x != y => l[x] != l[y]
```

Or, using disj, we could write all disj x, y instead and skip the condition.

Answer to Exercise 5 Property Testing Find

In python:

```
@given(s.lists(s.integers()), s.integers())
def test_myfind(l, x):
    out = myfind(l, x)
    if out == -1:
        assert x not in l
    else:
        assert l[out] == x
        assert x not in l[0:out]
```

Note that this will statistically overtest the case where x is not in l. Part of learning to use PBT well is getting a sense of how to best generate inputs. The techniques here are beyond the scope of this book.

Index