Design by Contract in Elixir

“Let it crash” meets “it shouldn’t crash” ride

#ONELESSCAR

Elba Sánchez Márquez
ride.com

@elbasanchezm

Guillermo Iguarán Suarez
ride.com

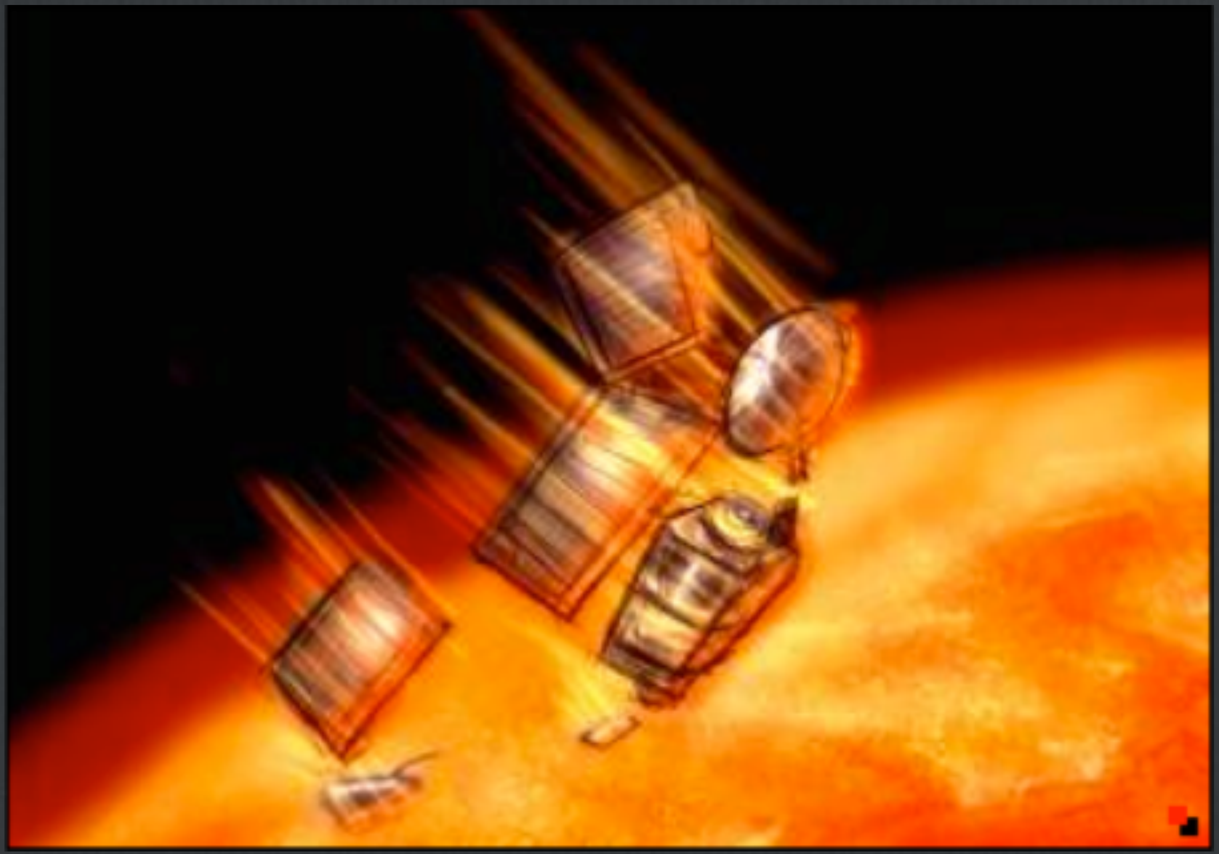
@guilleiguaran

Bugs and crashes

NOW WHAT DO I DO?
CLICK SOME BUTTONS AND SEE WHAT HAPPENS. YOU CAN'T HURT ANYTHING.
Dilbert.com DilbertCartoonist@gmail.com
CLICK
FZZEET!
10-15-09 ©2009 Scott Adams, Inc./Dist. by UFS, Inc.
IN MY DEFENSE, THAT HARDLY EVER HAPPENS.

Did you know?

There are really expensive software errors



NASA's Mars Climate Orbiter

~~non-SI~~ metric units ✓

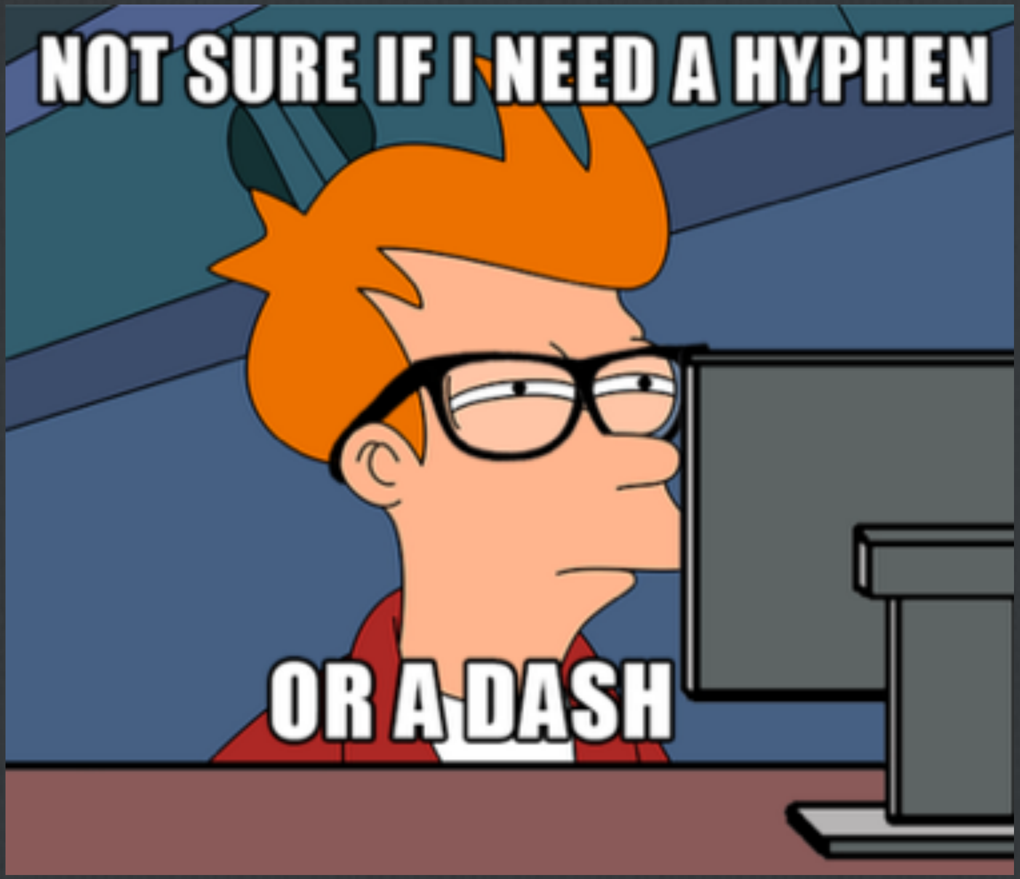
Did you know?

There are really expensive software errors

Heathrow Terminal 5 Opening

Did you know?

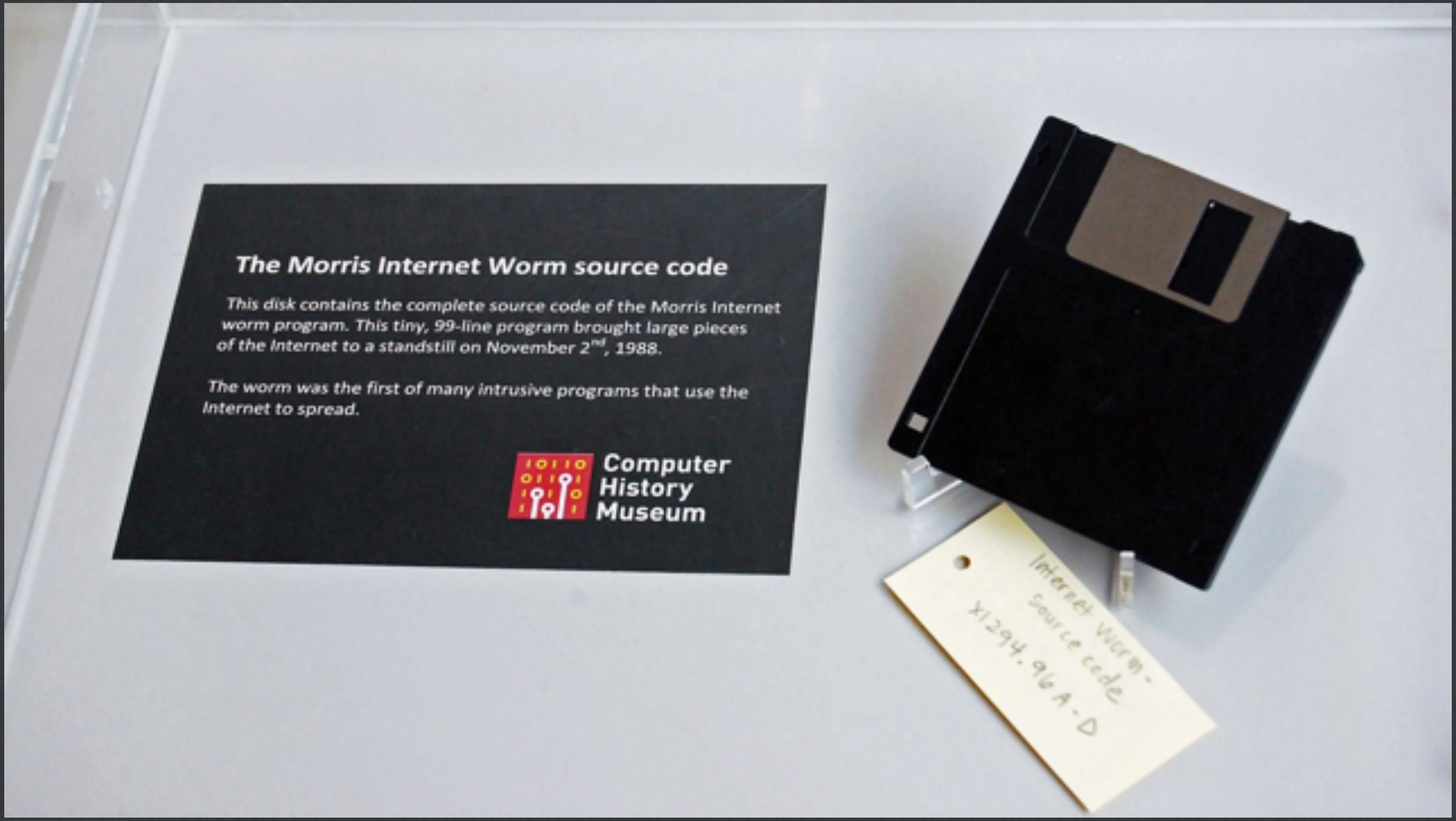
There are really expensive software errors



The Mariner 1 Spacecraft

Did you know?

There are really expensive software errors



The Morris Internet Worm source code

This disk contains the complete source code of the Morris Internet worm program. This tiny, 99-line program brought large pieces of the Internet to a standstill on November 2nd, 1988.

The worm was the first of many intrusive programs that use the Internet to spread.

Computer History Museum

The Morris worm

Did you know?

There are really expensive software errors

Ariane 5 flight 501

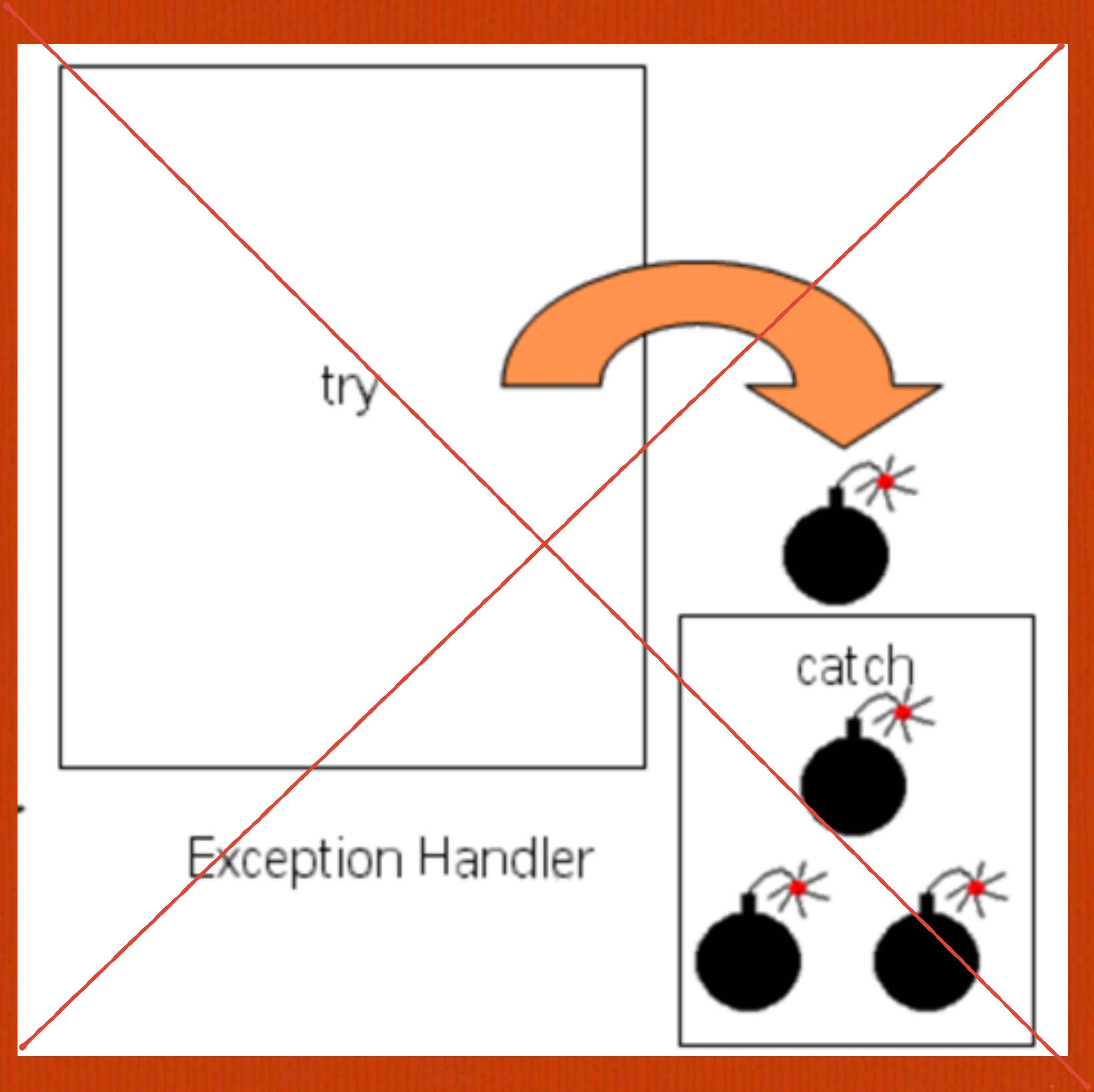
"Ariane 501 Cluster" by Phrd/de:user:Stahlkocher - Own work.
Licensed under CC BY-SA 3.0

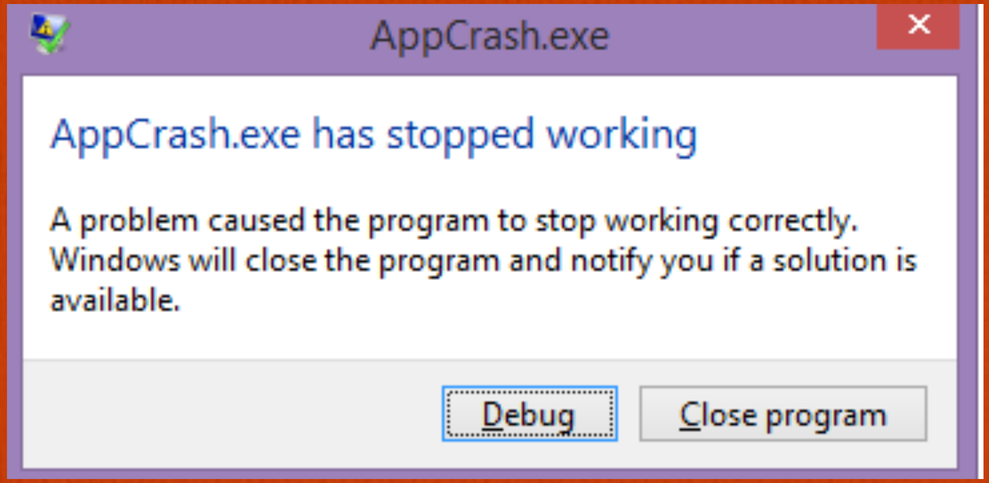
Its fastest engines exploited a bug that was not found in previous models

What happened?

the software had tried to cram a 64-bit number into a 16-bit space

What happened?





AppCrash.exe has stopped working

A problem caused the program to stop working correctly. Windows will close the program and notify you if a solution is available.

They couldn’t

Blame management

Blame the language

They couldn’t

Blame implementation

Blame testing

They blamed

the reuse specification

horizontal_bias <= maximum_bias

Back to design by contract…

A little bit of theory

History

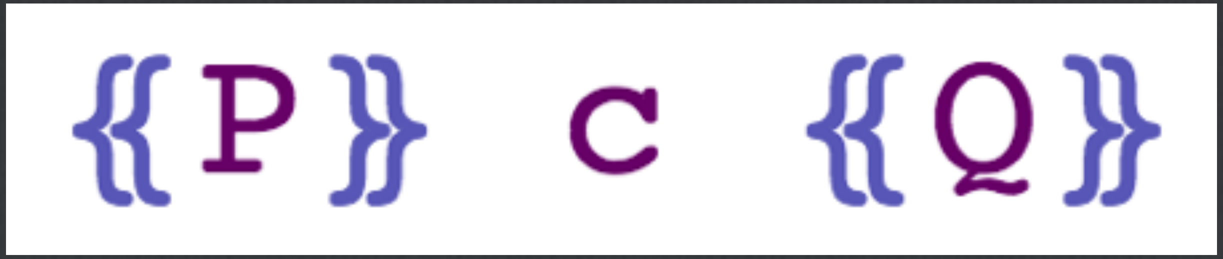
Design by contract has its roots in work on formal verification, formal specification and Hoare logic.

History

Hoare described the use of representation invariants and abstract functions to prove correctness of abstract data types

Basics of Hoare logic

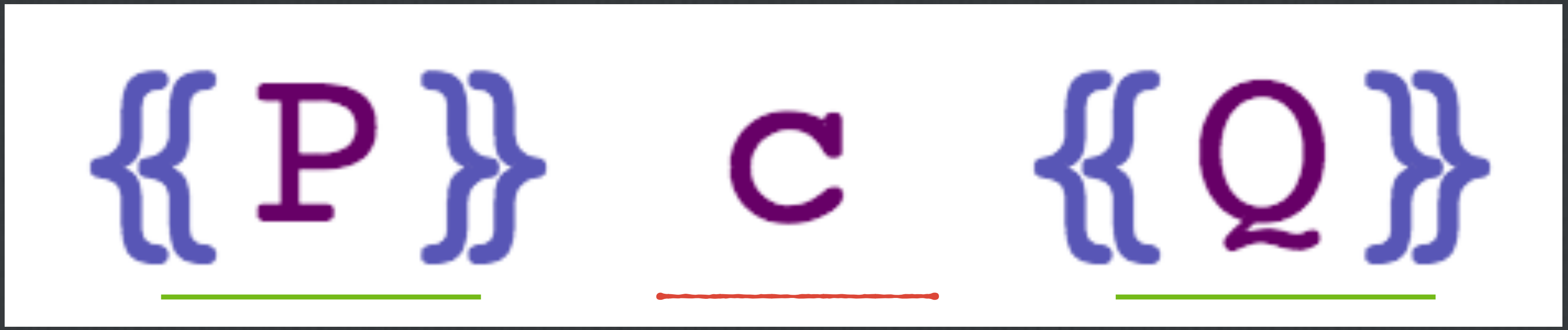
Hoare Logic is at the core of the deductive approach of the DbC.



{{P}} c {{Q}}

Formal reasoning about program correctness using pre and postconditions

Basics of Hoare logic



{{P}} c {{Q}}

Basics of hoare logic

- Provides axioms and inference rules
- There are rules for concurrency, procedures, jumps, and pointers.

DbC vs. Testing

Design by contract (DbC)

- Software correctness methodology
- Programmatically asserts the change in state caused by a piece of a program

Unit tests

- Used to verify that the software works correctly
- Hard to detect all possible edge cases during development.

What is Design by Contract?

	Obligations	Rights
Passenger	Buy Airline ticket, bring accepted baggage and be at airport 2 hours before	Reach destination
Airline	Bring passenger to destination	No need to carry passenger who is late, or has unacceptable baggage, or hasn't paid ticket

- □ Each party benefits and accepts obligations
- □ One party's benefits are the other party's obligation
- □ It is described so that both parties understand what would be guaranteed without saying how.

Structure of a Contract

Structure of a contract

PRECONDITION

Requires clause

Structure of a contract

POSTCONDITION

Ensures Clause

Structure of a contract

IF PRECONDITION TRUE

→ EXECUTE

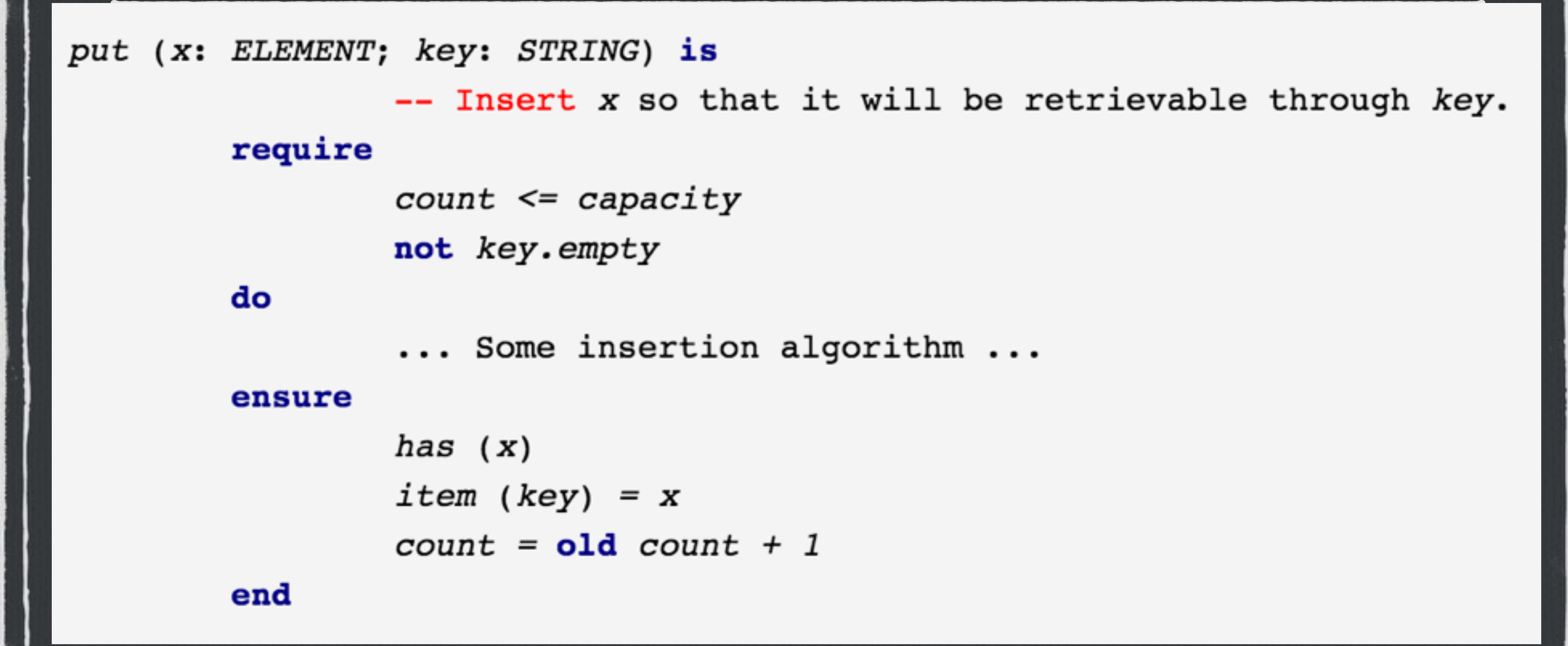
→ POSTCONDITION TRUE

Structure of a contract

IF PRECONDITION FALSE

→ NOTHING

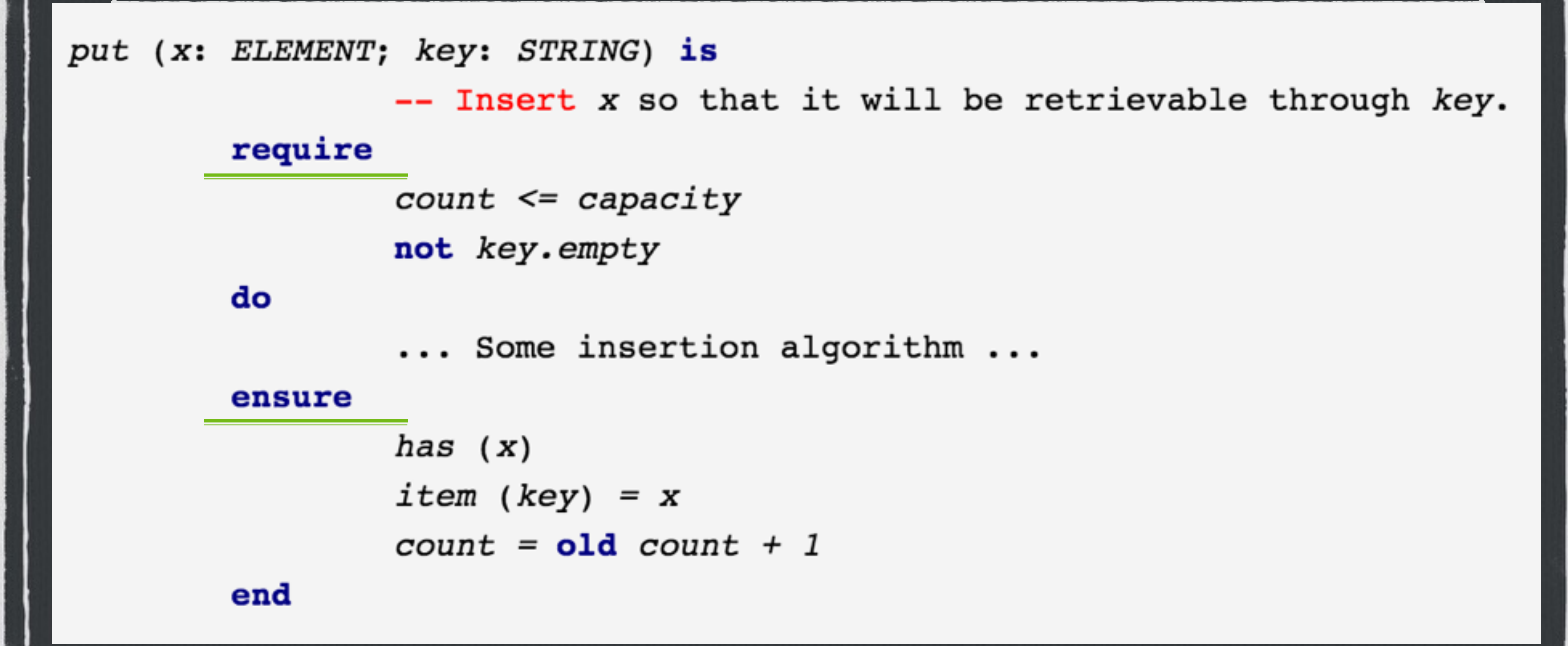
Example



```
put (x: ELEMENT; key: STRING) is
            -- Insert x so that it will be retrievable through key.
      require
            count <= capacity
            not key.empty
      do
            ... Some insertion algorithm ...
      ensure
            has (x)
            item (key) = x
            count = old count + 1
      end
```

https://www.eiffel.com/values/design-by-contract/introduction/

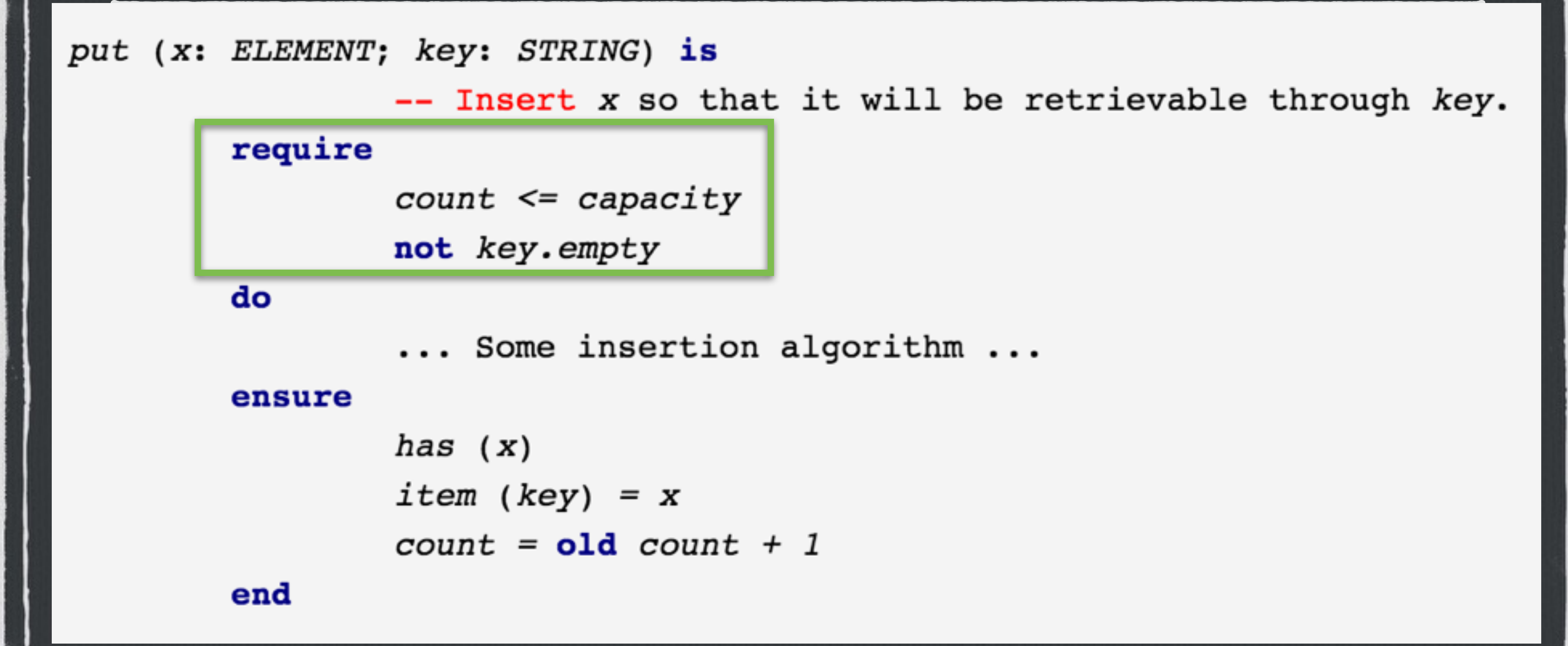
Example



```
put (x: ELEMENT; key: STRING) is
                -- Insert x so that it will be retrievable through key.
        require
                count <= capacity
                not key.empty
        do
                ... Some insertion algorithm ...
        ensure
                has (x)
                item (key) = x
                count = old count + 1
        end
```

https://www.eiffel.com/values/design-by-contract/introduction/

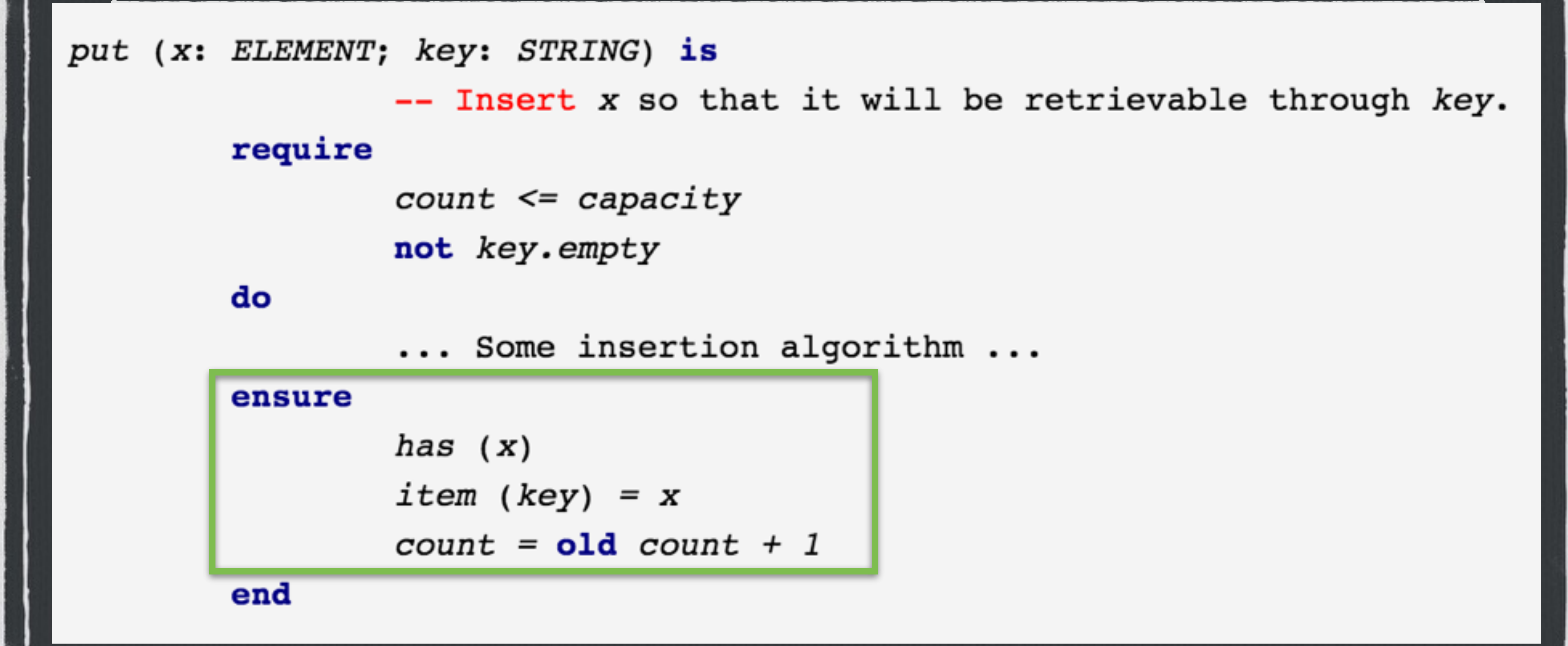
Example



```
put (x: ELEMENT; key: STRING) is
                -- Insert x so that it will be retrievable through key.
        require
                count <= capacity
                not key.empty
        do
                ... Some insertion algorithm ...
        ensure
                has (x)
                item (key) = x
                count = old count + 1
        end
```

https://www.eiffel.com/values/design-by-contract/introduction/

Example

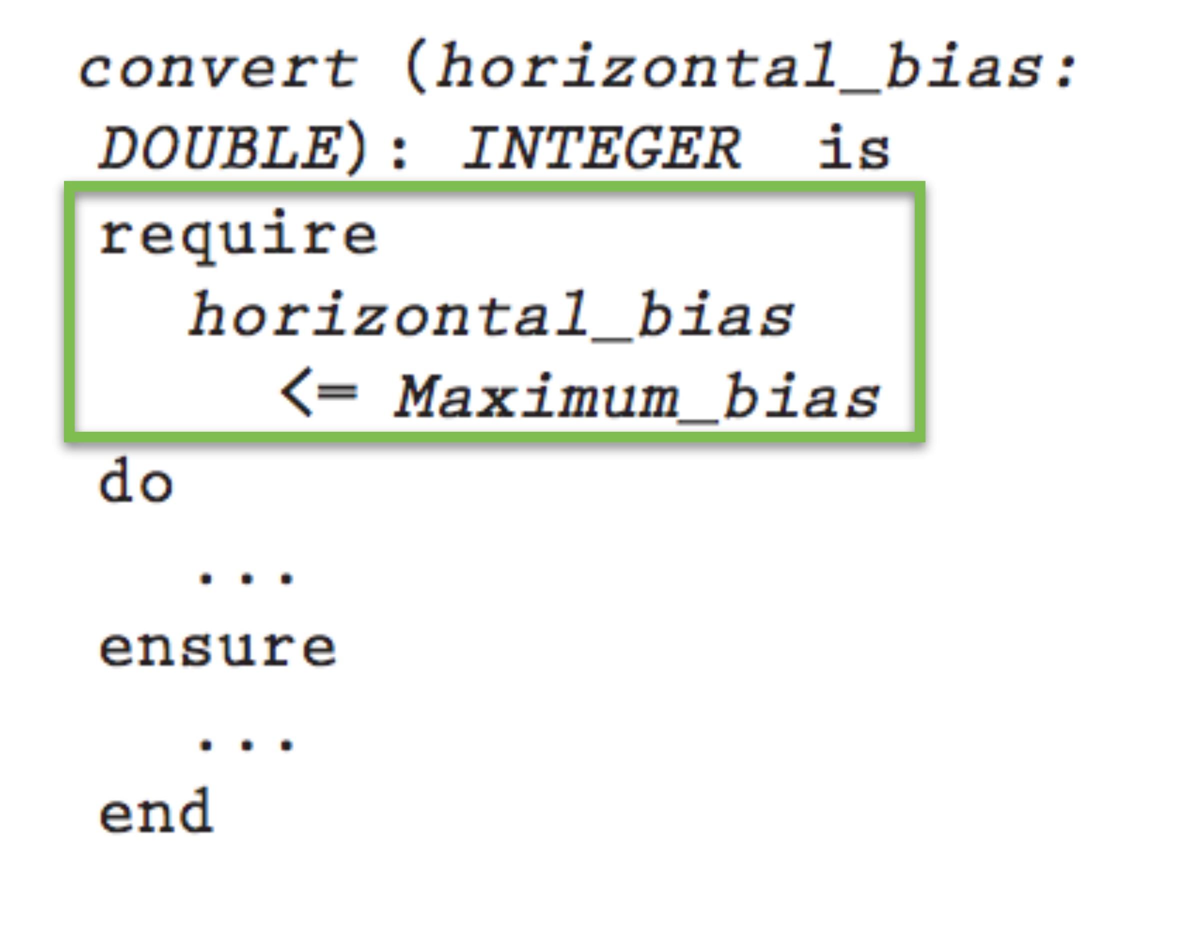


```
put (x: ELEMENT; key: STRING) is
                -- Insert x so that it will be retrievable through key.
        require
                count <= capacity
                not key.empty
        do
                ... Some insertion algorithm ...
        ensure
                has (x)
                item (key) = x
                count = old count + 1
        end
```

https://www.eiffel.com/values/design-by-contract/introduction/

in Ariane's case

Where the precondition (require…) states clearly and precisely what the input must satisfy to be acceptable.



```
convert (horizontal_bias:
 DOUBLE): INTEGER  is
 require
    horizontal_bias
       <= Maximum_bias
 do
    ...
 ensure
    ...
 end
```

Eiffel Version

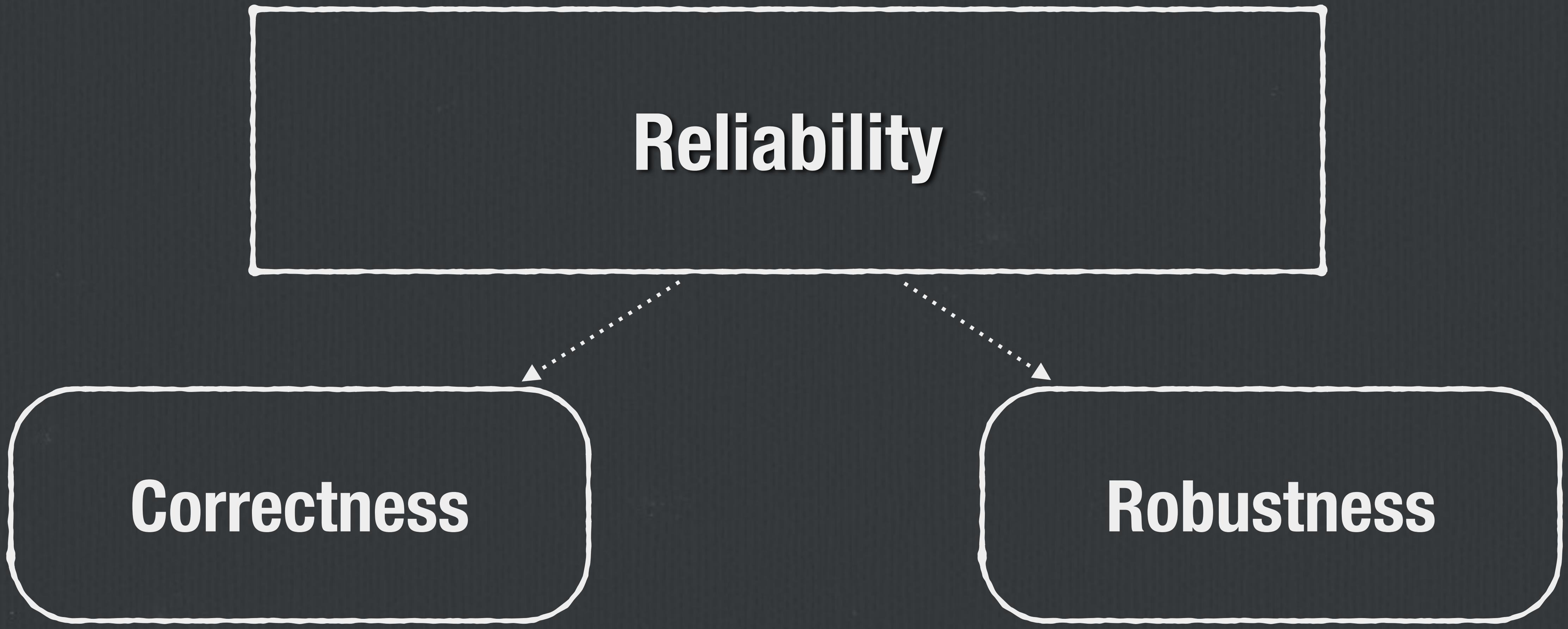
how can this help?

> “When quality is pursued,
> productivity follows.”

–K. Fujino

Vice President of NEC Corporation’s C&C Software Development Group

And also we look for…



Reliability

Correctness

Robustness

Advantages

- Assertions (preconditions and postconditions in particular) can be automatically turned on during testing.

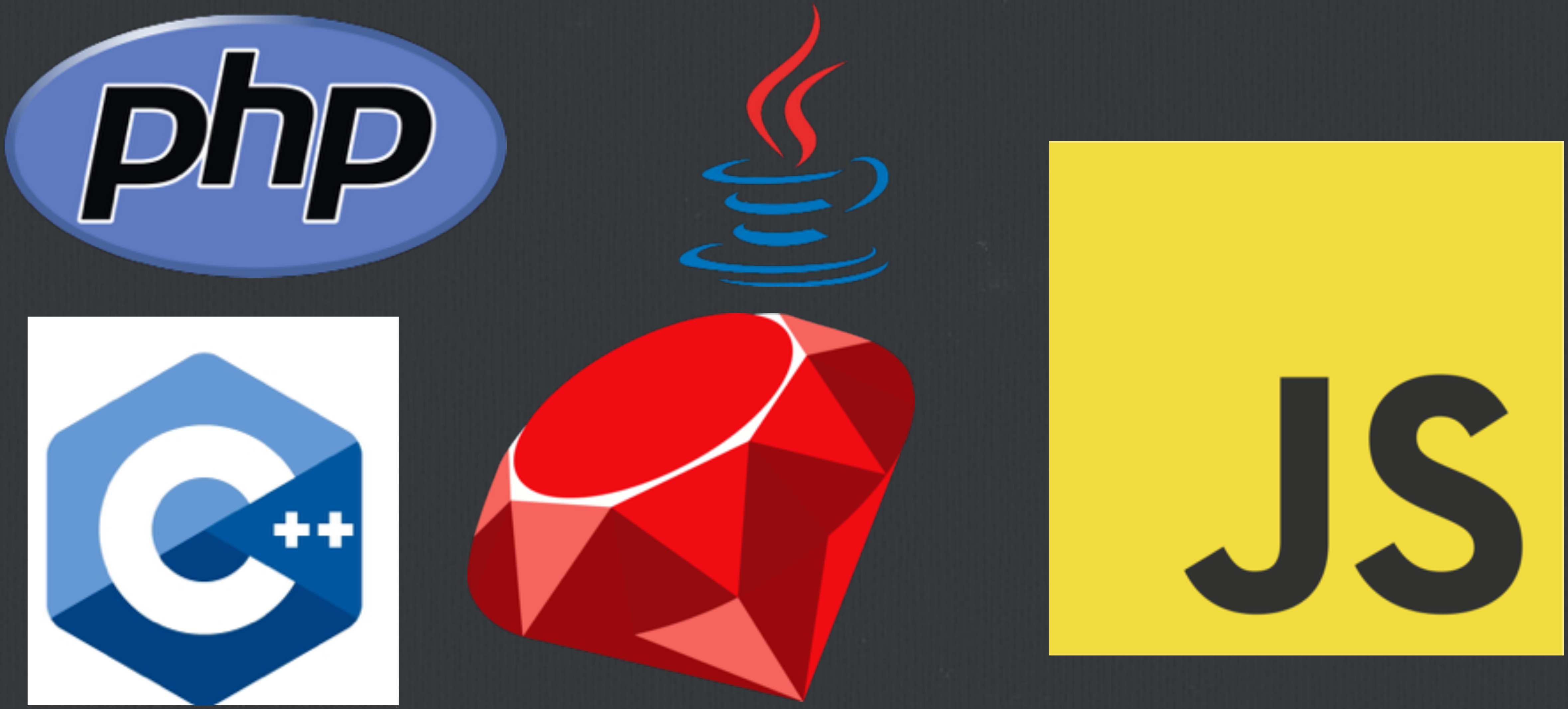
Advantages

- Assertions can remain turned on during execution, triggering an exception if violated.

Advantages

- Assertions are a prime component of the software and its automatically produced documentation.

Language support



There are several implementations of DbC libraries for some languages

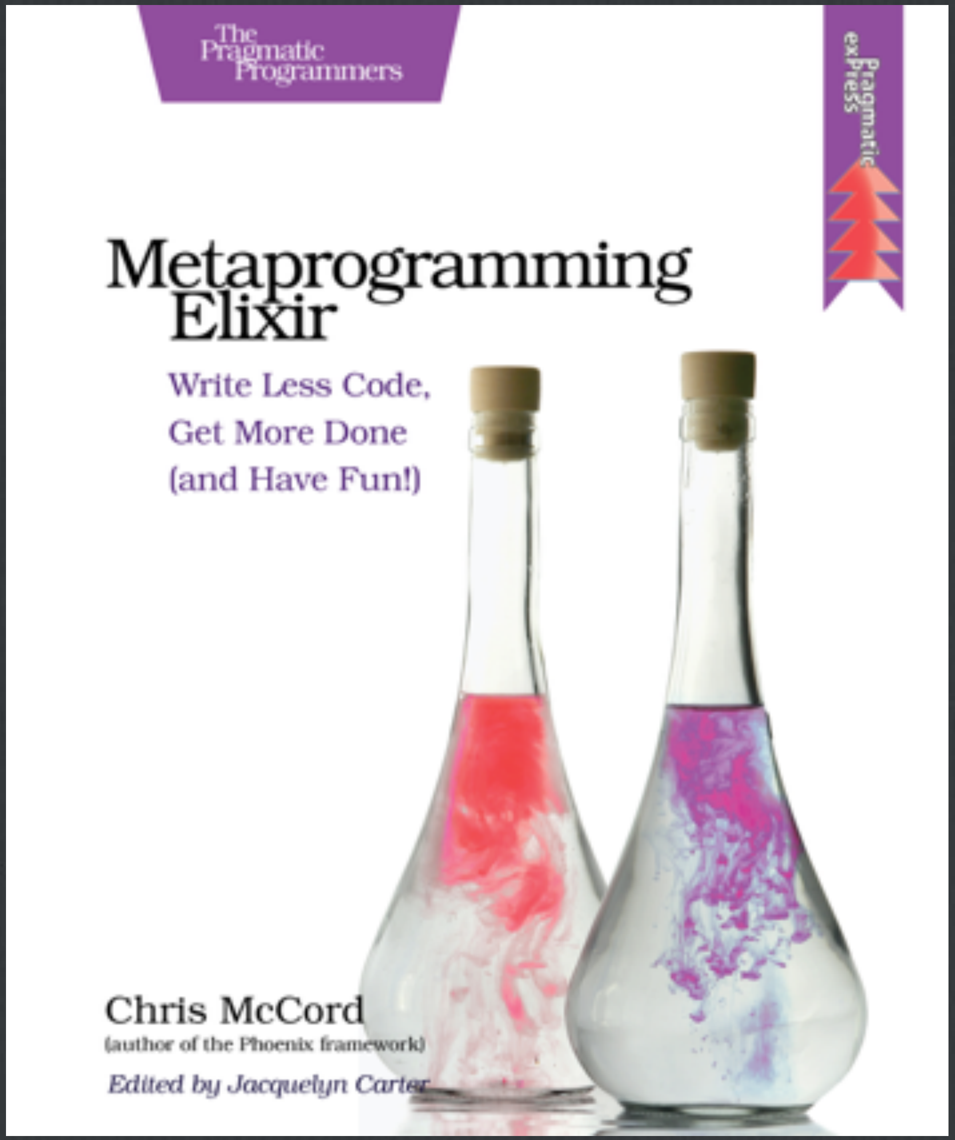
Language support

And languages with native support

Eiffel
Software

and now with Elixir

Metaprogramming In Elixir


The Pragmatic Programmers
Pragmatic exPress
Metaprogramming Elixir
Write Less Code, Get More Done (and Have Fun!)
Chris McCord
(author of the Phoenix framework)
Edited by Jacquelyn Carter

Book by Chris McCord - O'Reilly

Macros

Macros Rules

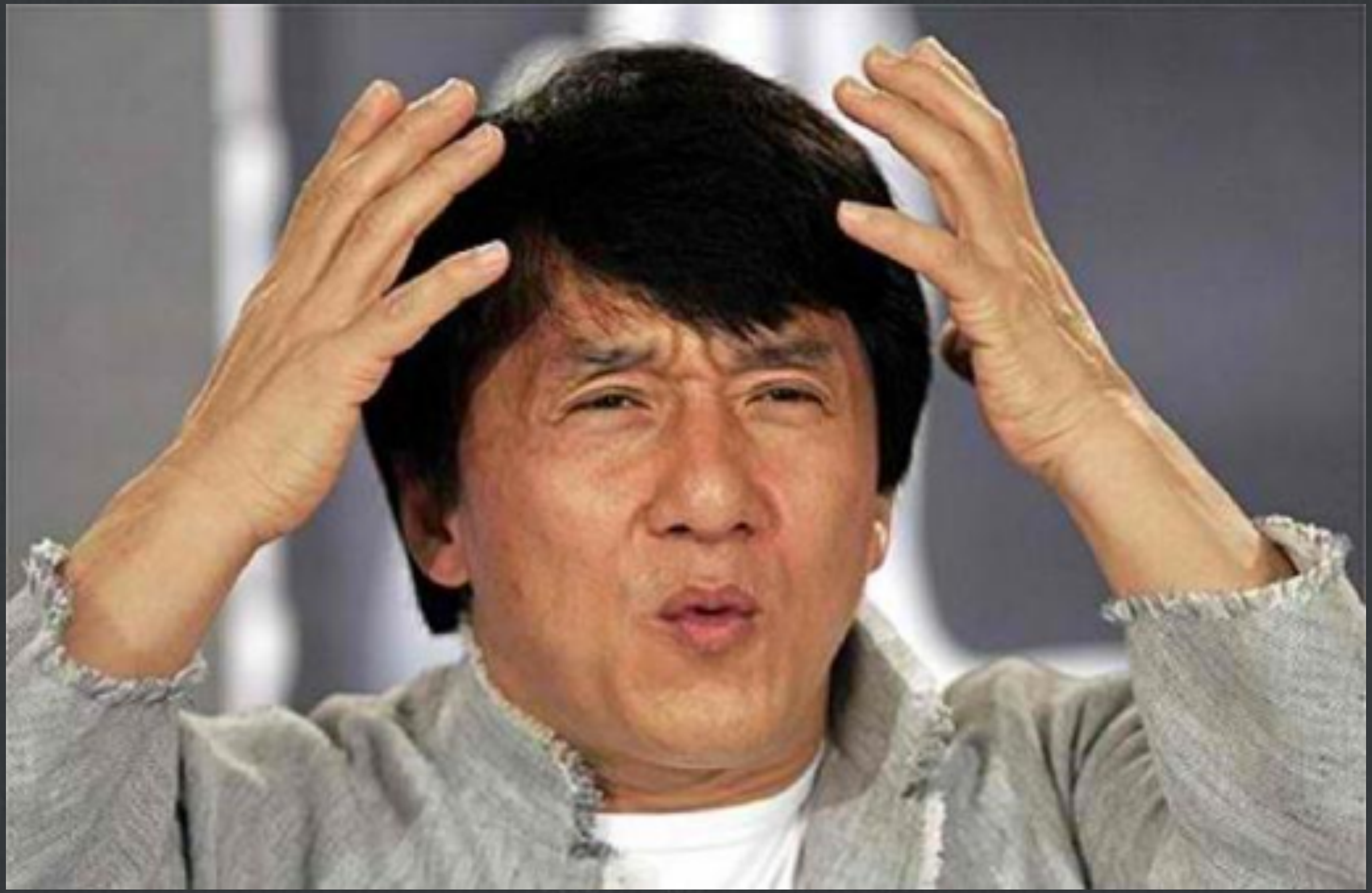
Rule #1

Don’t write Macros

Macros Rules

Rule #2

Use Macros gratuitously

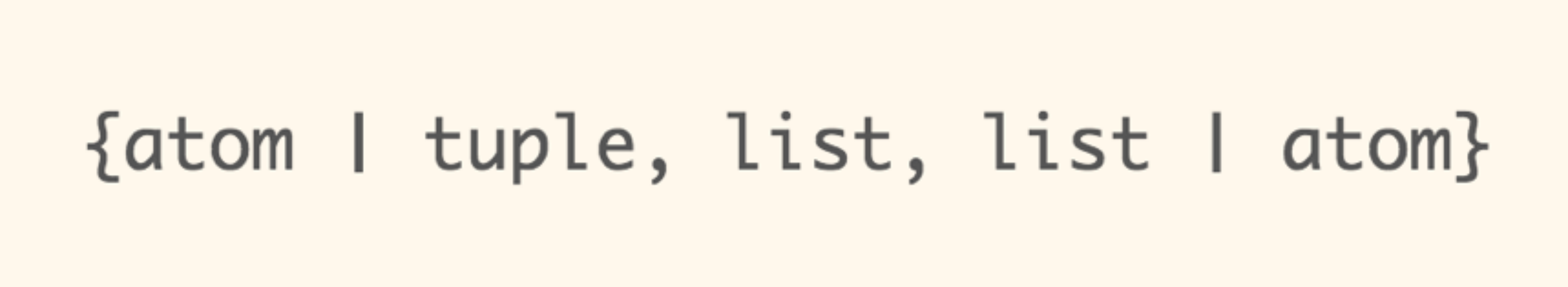


Macros

- A macro is code that writes code
- Many constructs in Elixir are macros (def, if, unless, defmodule)
- Elixir code runs at compile time and can be used to manipulate language AST.

Macros

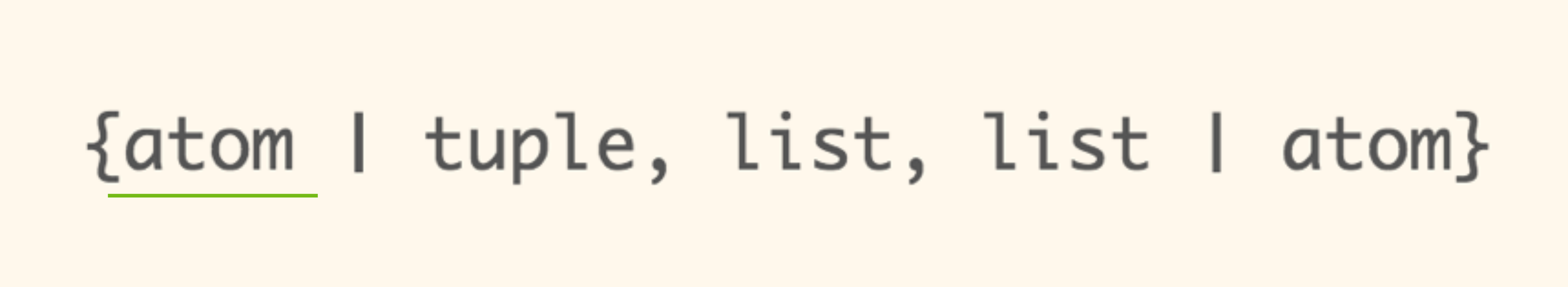
Abstract Syntax Tree



```
{atom | tuple, list, list | atom}
```

Macros

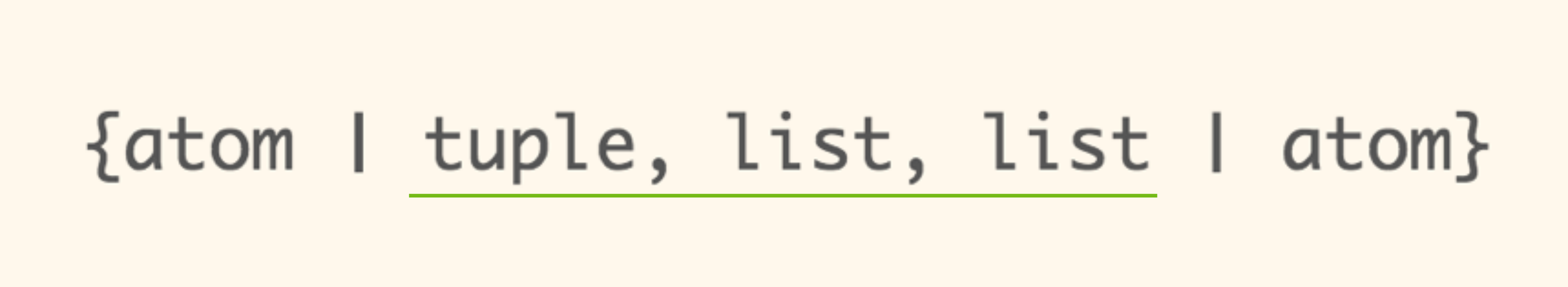
Abstract Syntax Tree



```
{atom | tuple, list, list | atom}
```

Macros

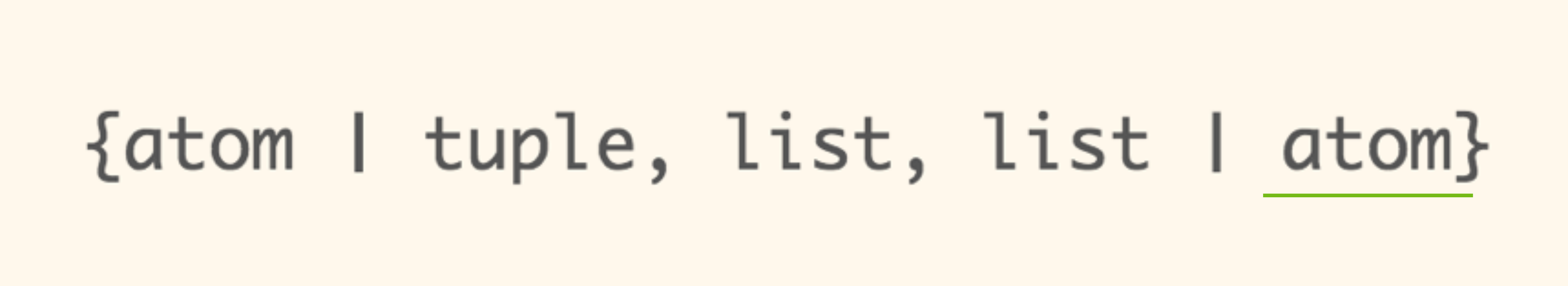
Abstract Syntax Tree



```
{atom | tuple, list, list | atom}
```

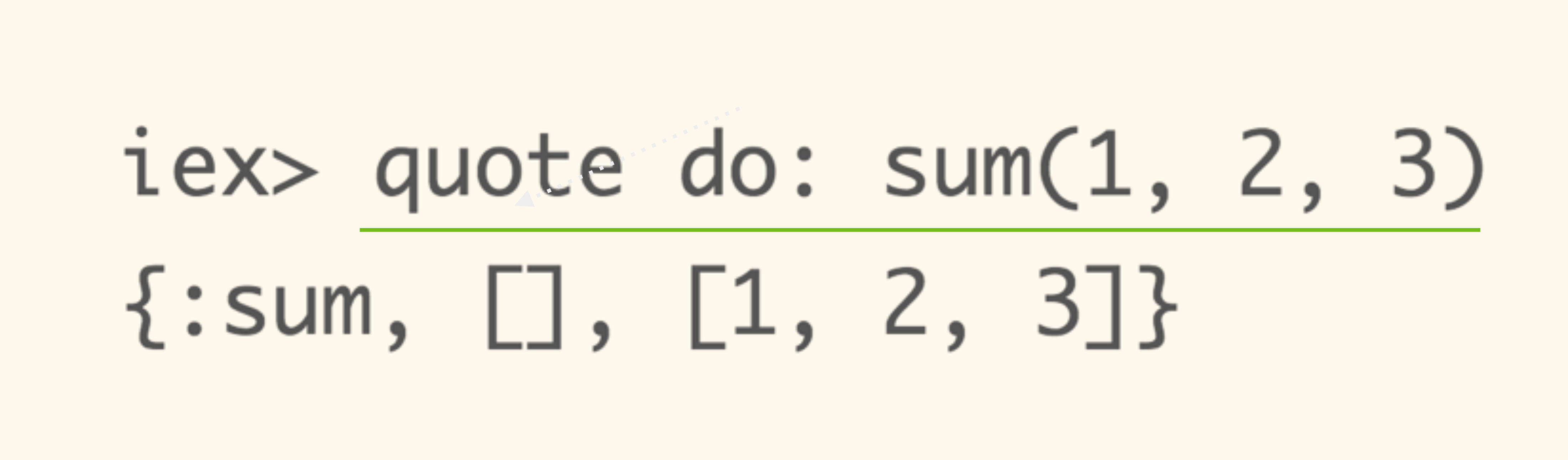
Macros

Abstract Syntax Tree



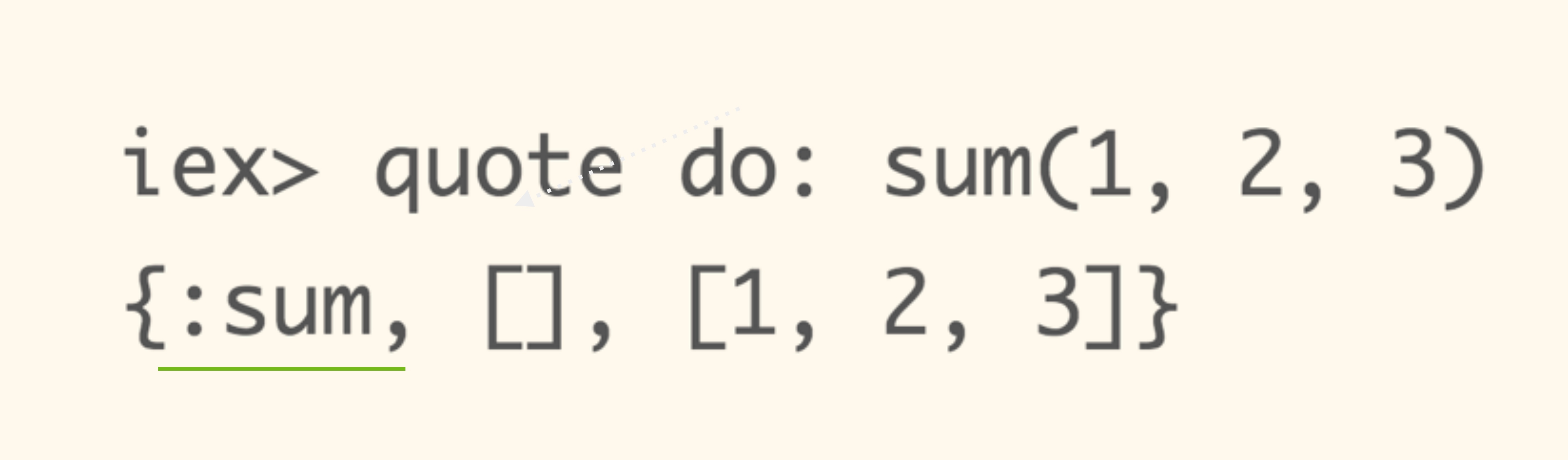
```
{atom | tuple, list, list | atom}
```

Quote Macro



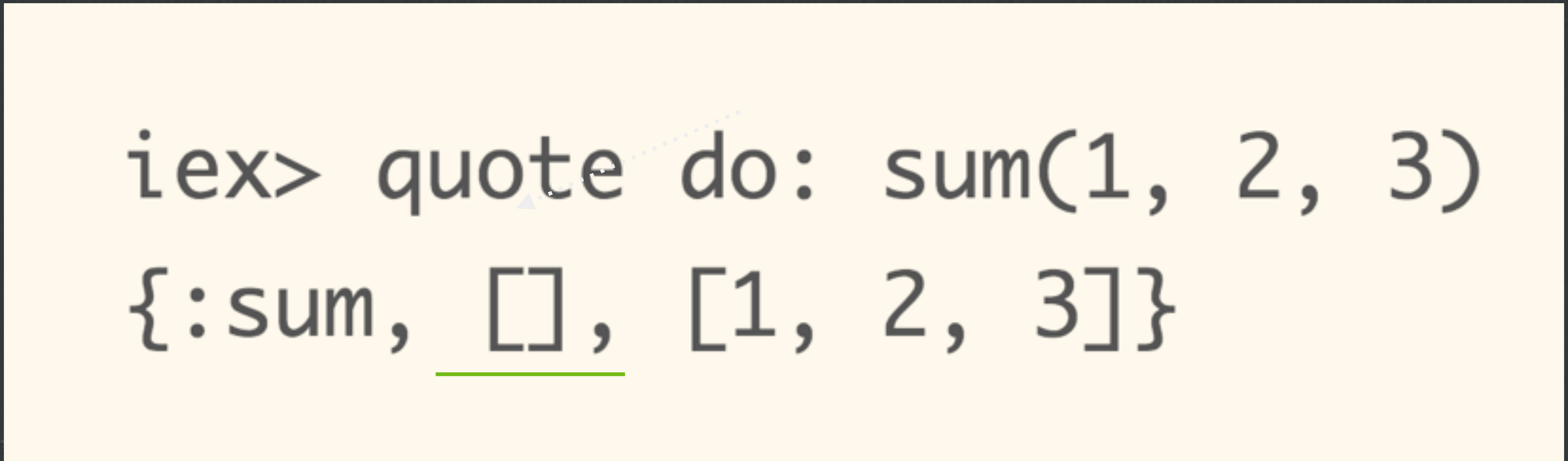
```
iex> quote do: sum(1, 2, 3)
{:sum, [], [1, 2, 3]}
```

Quote Macro



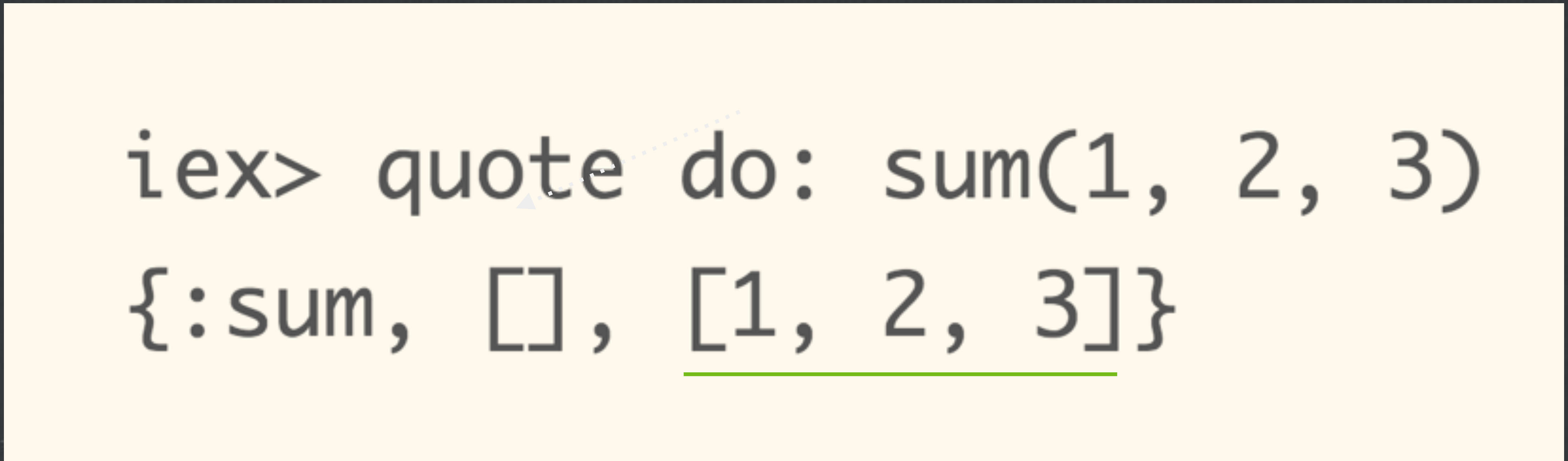
```
iex> quote do: sum(1, 2, 3)
{:sum, [], [1, 2, 3]}
```

Quote Macro



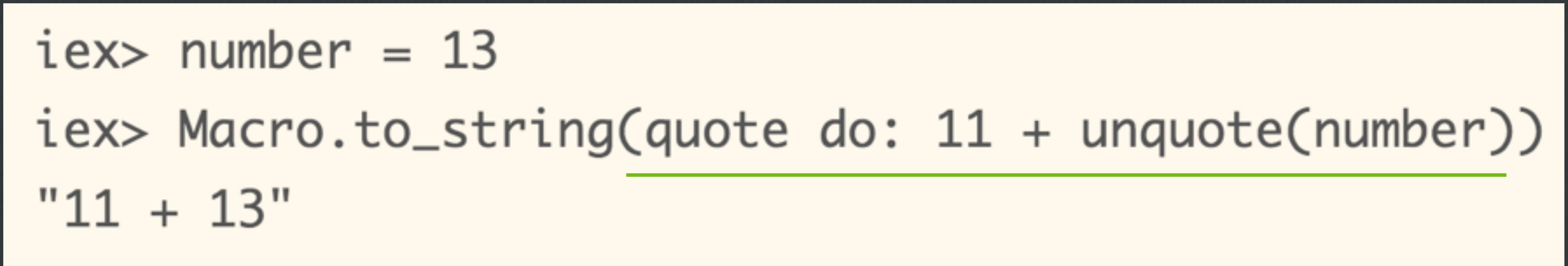
```
iex> quote do: sum(1, 2, 3)
{:sum, [], [1, 2, 3]}
```

Quote Macro



```
iex> quote do: sum(1, 2, 3)
{:sum, [], [1, 2, 3]}
```

Unquote Macro

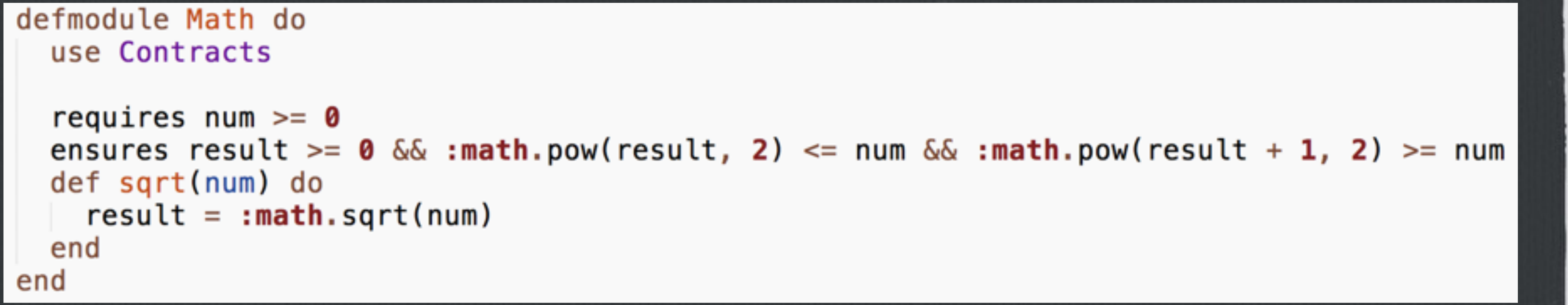


```
iex> number = 13
iex> Macro.to_string(quote do: 11 + unquote(number))
"11 + 13"
```

Back to DbC

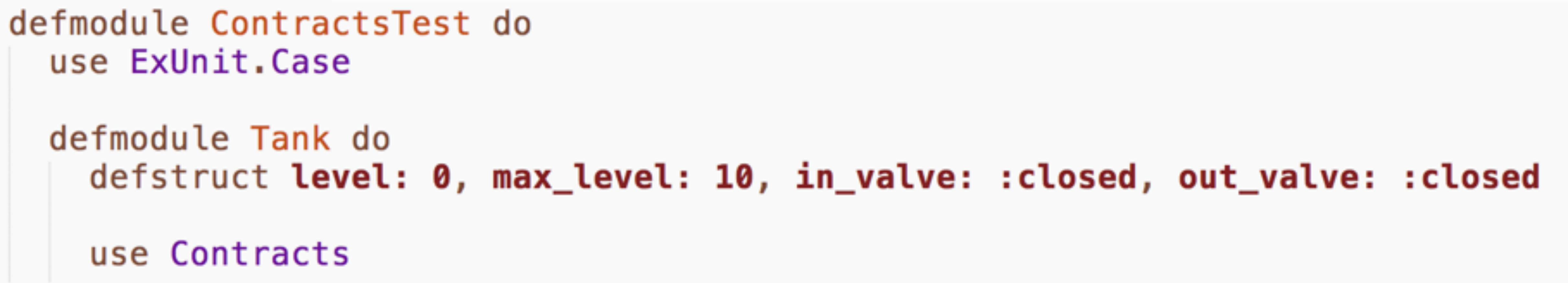
- We used Elixir macros to extend the language adding support for basic DbC constructs.
- We tagged existing functions with “requires” and “ensures” tags.
- Macros manipulate function body to insert precondition and postconditions inside of functions.

What we had to do



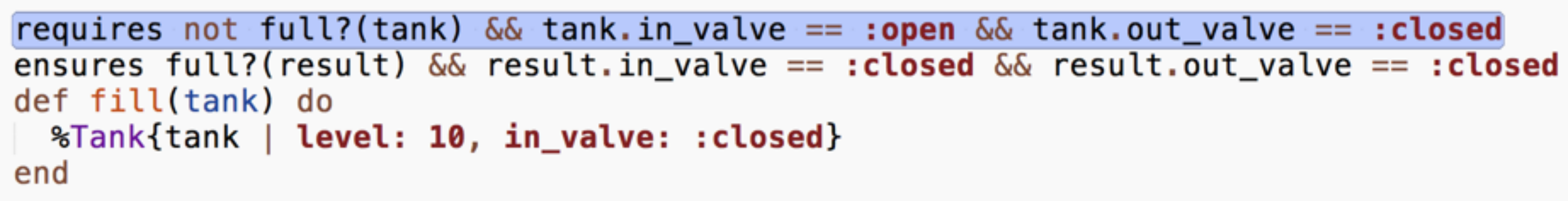
```
defmodule Math do
  use Contracts

  requires num >= 0
  ensures result >= 0 && :math.pow(result, 2) <= num && :math.pow(result + 1, 2) >= num
  def sqrt(num) do
    result = :math.sqrt(num)
  end
end
```

demo



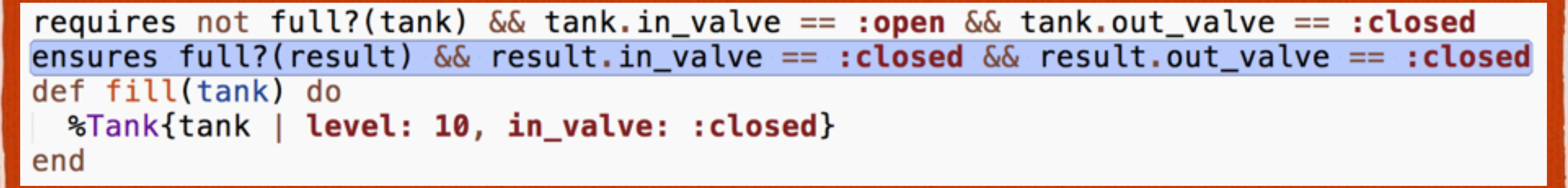
```
defmodule ContractsTest do
  use ExUnit.Case

  defmodule Tank do
    defstruct level: 0, max_level: 10, in_valve: :closed, out_valve: :closed

    use Contracts
```

Precondition

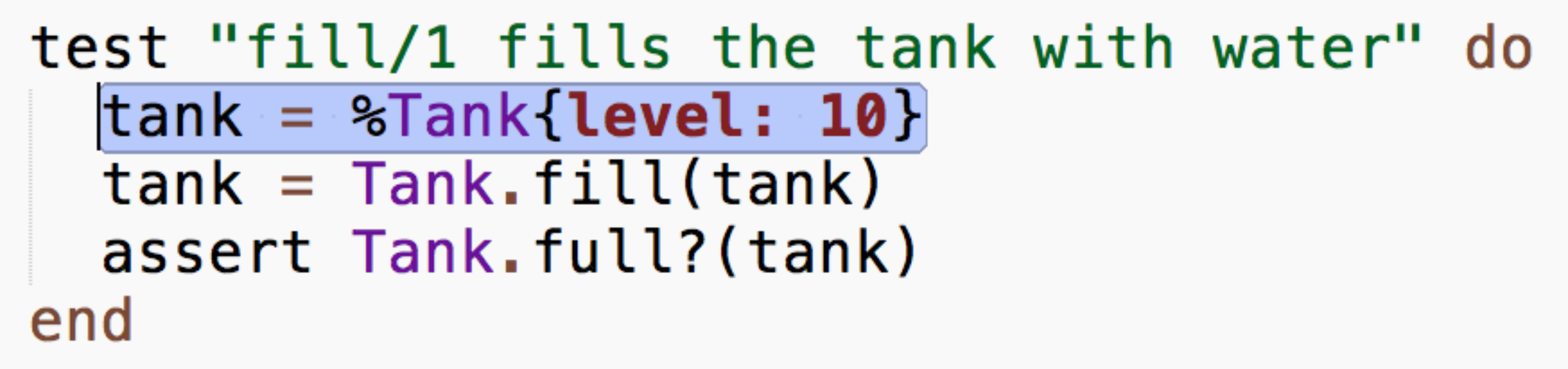


```
requires not full?(tank) && tank.in_valve == :open && tank.out_valve == :closed
ensures full?(result) && result.in_valve == :closed && result.out_valve == :closed
def fill(tank) do
  %Tank{tank | level: 10, in_valve: :closed}
end
```

Postcondition

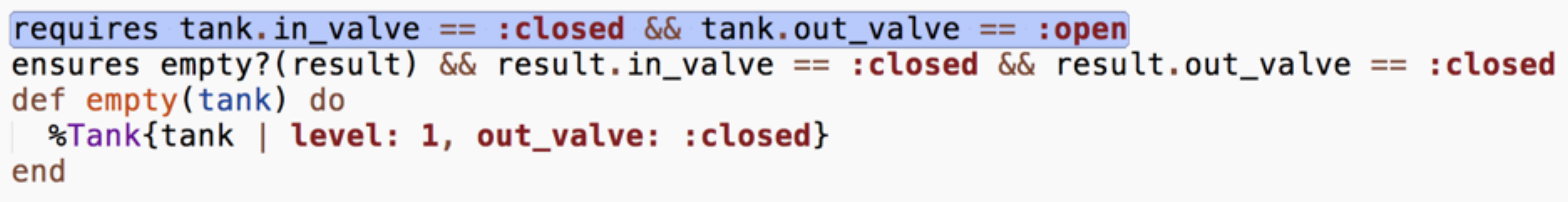


```
requires not full?(tank) && tank.in_valve == :open && tank.out_valve == :closed
ensures full?(result) && result.in_valve == :closed && result.out_valve == :closed
def fill(tank) do
  %Tank{tank | level: 10, in_valve: :closed}
end
```



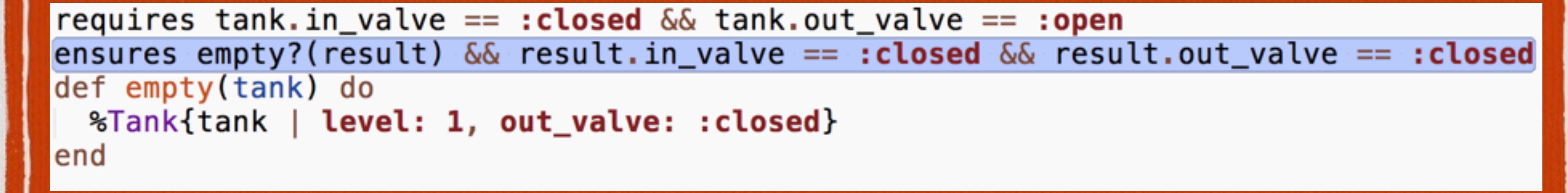
```
test "fill/1 fills the tank with water" do
  tank = %Tank{level: 10}
  tank = Tank.fill(tank)
  assert Tank.full?(tank)
end
```

Precondition



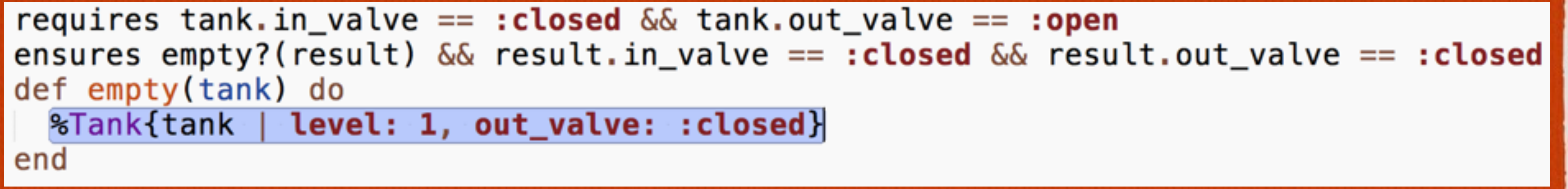
```
requires tank.in_valve == :closed && tank.out_valve == :open
ensures empty?(result) && result.in_valve == :closed && result.out_valve == :closed
def empty(tank) do
  %Tank{tank | level: 1, out_valve: :closed}
end
```

Postcondition

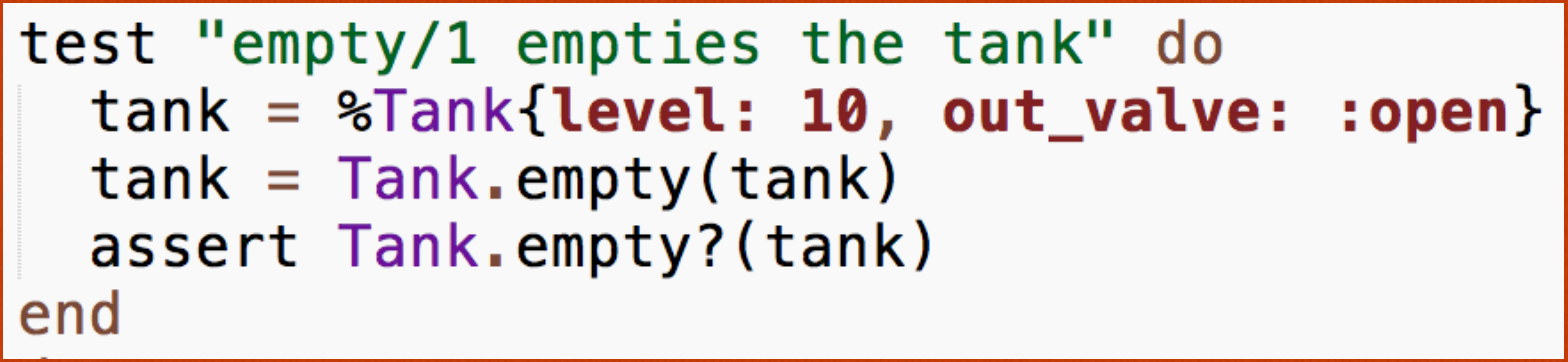


```
requires tank.in_valve == :closed && tank.out_valve == :open
ensures empty?(result) && result.in_valve == :closed && result.out_valve == :closed
def empty(tank) do
  %Tank{tank | level: 1, out_valve: :closed}
end
```

Command



```
requires tank.in_valve == :closed && tank.out_valve == :open
ensures empty?(result) && result.in_valve == :closed && result.out_valve == :closed
def empty(tank) do
  %Tank{tank | level: 1, out_valve: :closed}
end
```



```
test "empty/1 empties the tank" do
  tank = %Tank{level: 10, out_valve: :open}
  tank = Tank.empty(tank)
  assert Tank.empty?(tank)
end
```

Github: epsanchezma elixir-contracts

https://goo.gl/5f9GiU

FURTHER WORK

- □ Generate test-cases from Contracts
- □ Add configuration options to turn-on/off contracts in development and production
- □ Generate automated documentation from contracts
- □ Generate QuickCheck tests

To conclude

- □ Design by contract does not replace regular testing strategies
- □ Contracts add an extra grade of reliability
- □ It’s not a silver bullet

References

- □ Ariane’s case: http://se.inf.ethz.ch/~meyer/publications/computer/ariane.pdf
- □ DbC History: http://c2.com/cgi/wiki?DesignByContract
- □ Hoare Logic: https://www.cs.cmu.edu/~aldrich/courses/654-sp07/slides/7-hoare.pdf
- □ DbC: http://ansymore.uantwerpen.be/system/files/uploads/courses/SE3BAC/06DesignContract.pdf, http://web.cse.ohio-state.edu/software/2221/web-sw1/extras/slides/09.Design-by-Contract.pdf
- □ Examples: https://www.eiffel.com/

Thank you!