Functions… kinda like $f(x)$

CMPU 101 – Problem Solving and Abstraction

Peter Lemieszewski

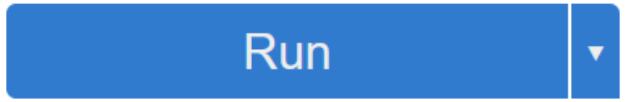
A quick review of last week's concepts

- We've been using Pyret to write expressions that use:
  1. Data, including numbers (0, -10, 0.4),
  2. strings ("" , "hi", "111"),
  3. images (circle(2, "solid", "red")).
- Which we modify or combine using operators or functions
  1. `+`
  2. `string-append`
  3. `overlay`

A quick review of last week's concepts

- We've been using Pyret to write expressions that use:
  1. Data, including numbers (0, -10, 0.4),
  2. strings ("" , "hi", "111"),
  3. images (circle(2, "solid", "red")).
- Which we modify or combine using operators or functions
  1. `+` Operator
  2. `string-append` Function
  3. `overlay` Function

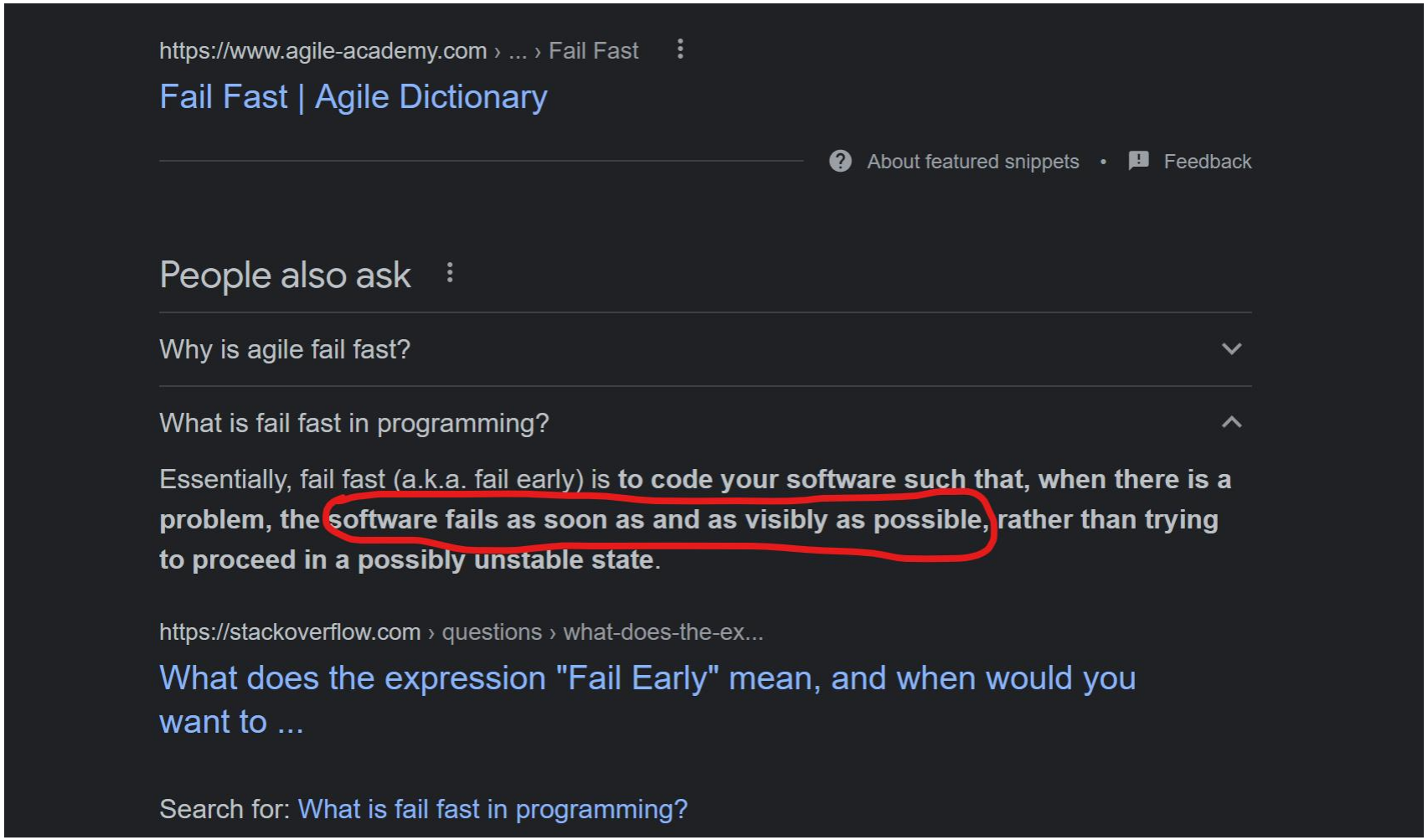
Errrors

- IRL Software Problems are expensive when found by customers
  - In terms of cost of lost revenue, fixing the problem, brand satisfaction
  - Typically called “run time errors”
  - Less expensive if found before released to customers
- The cost of a problem is lower… the earlier a problem is discovered
- Lowest cost: finding a problem immediately after we write it!
  - This is what pyret does as you type code
  - This is also what happens when you click on 

- In Agile programming terms: Fail Fast

Fail Fast

- …google search:



What else did we observe last week?

- We can create (more) sophisticated code by combining functions and operators
- …essentially creating expressions
  - 1 + (7 / 8)
  - string-append(“Computer”, “Science”)
- We can name our expressions too
- …more precisely, the *results* of our expressions
  - my-major = string-append(“Computer”, “Science”)

Today's Topic: Functions

- In mathematics:
  - $f(x) = x^3 + 2x + 1$
- In Pyret Programming:
  - We need a way to tell Pyret we have a `fun`ction:

```
fun
 f(x): = (x * x * x) + (2 * x) + 1
end
```

Pyret Function Syntax

Function definitions in Pyret have this form:

```
fun ⟨function-name⟩ (⟨arg-name⟩, ...):
  ⟨expression⟩
end
```

- Angle brackets < > refer to something that is optional
  - Technically, the ellipse should have angle brackets too!
- Another name for arg-name is parameter

SSE: Super Simple Example

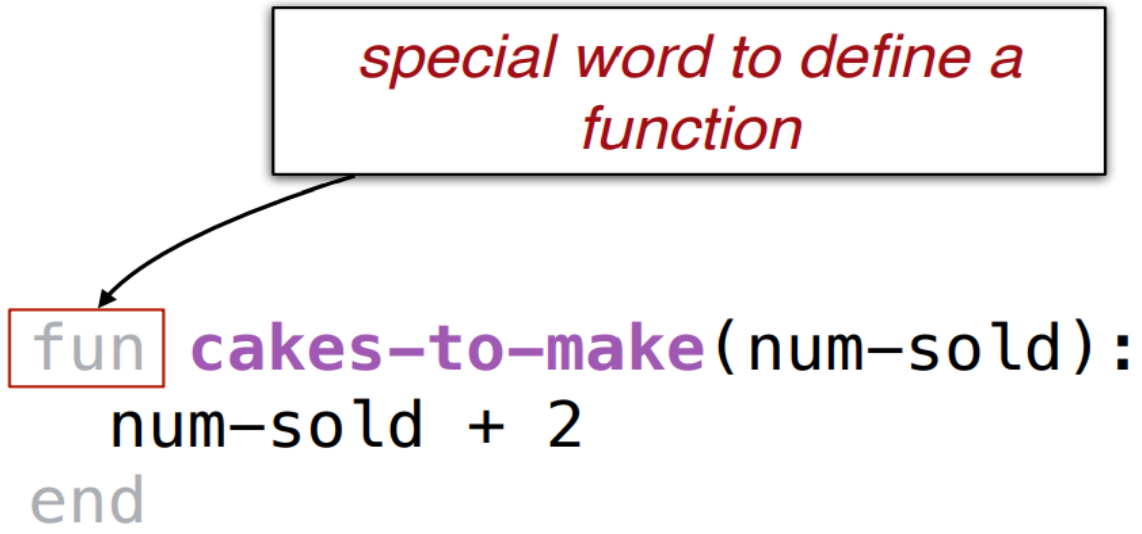
Mary Berry needs to know how many cakes to bake for her cake shop.

To avoid running out or having too many, she likes to bake two cakes more than the number she sold the previous day.

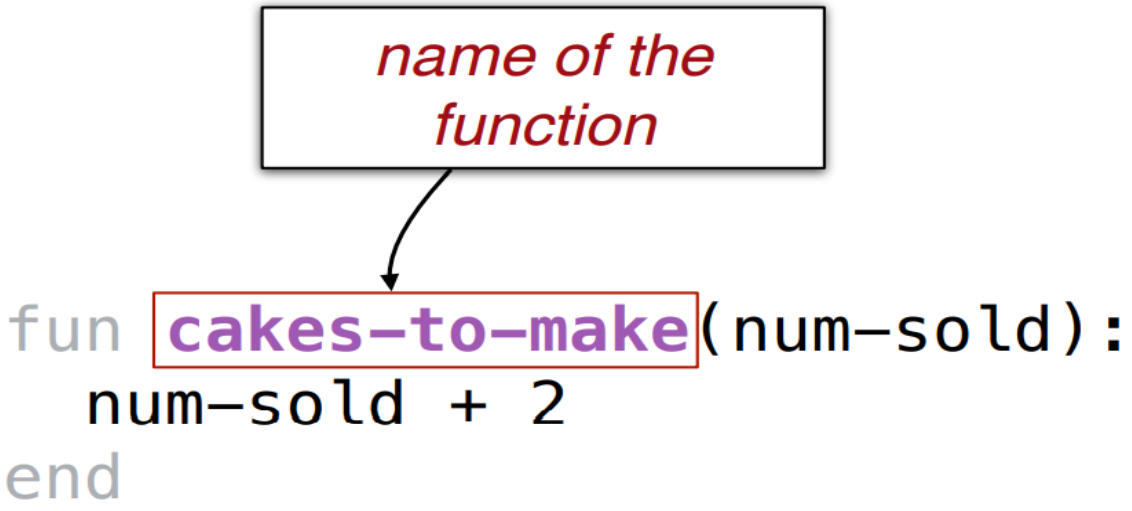
E.g., if Mary sells eight cakes on Monday, she makes ten cakes on Tuesday.

Let's write some code to help Mary.

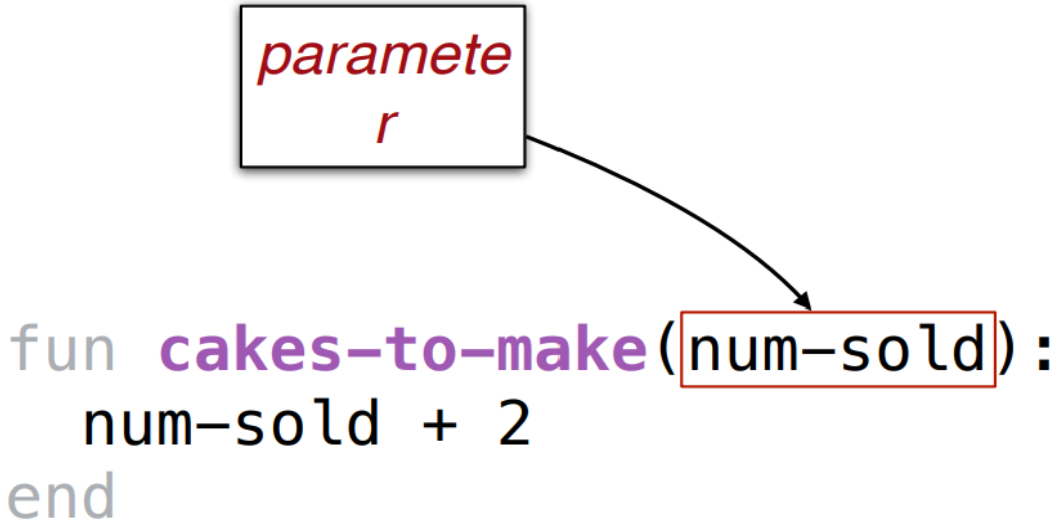
SSE: Pyret function elements 1/5



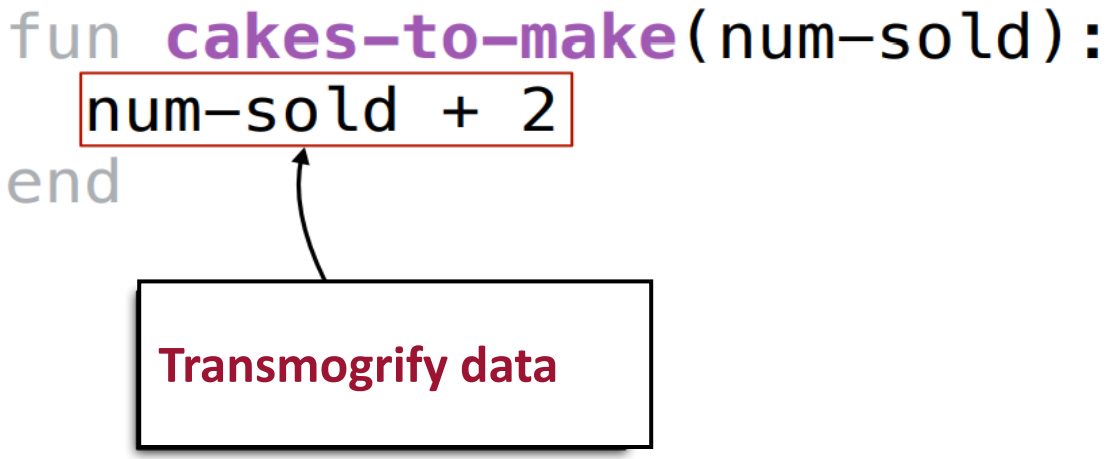
SSE: Pyret function elements 2/5



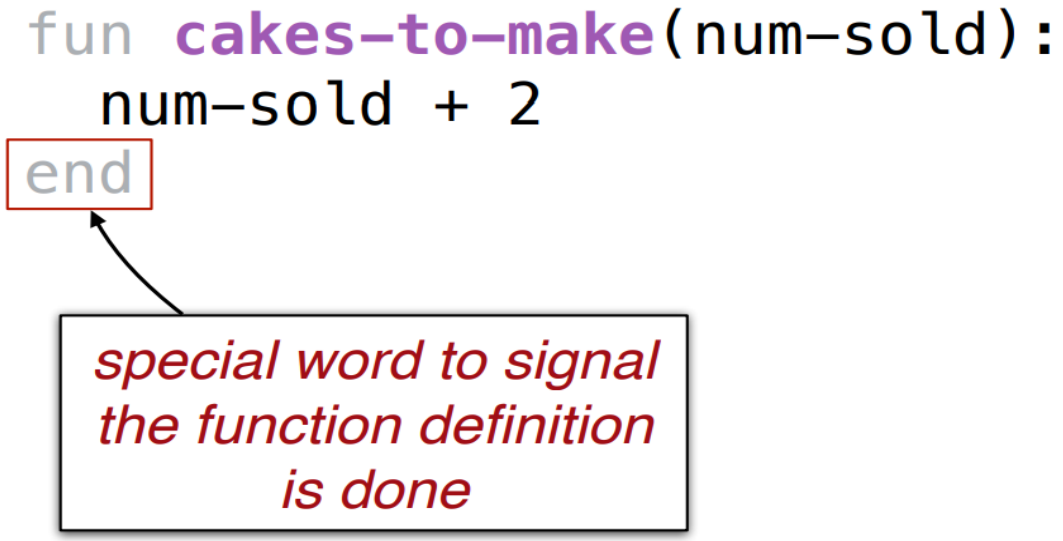
SSE: Pyret function elements 3/5



SSE: Pyret function elements 4/5



SSE: Pyret function elements 5/5

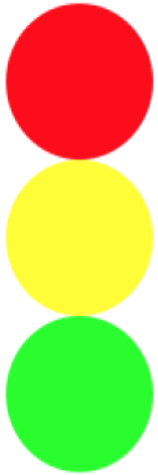


Functional Abstraction 1/3

```
# Draw a traffic light
above(  circle(40, "solid", "red"),
  above(circle(40, "solid", "yellow"),
        circle(40, "solid", "green"))
```

Unchanging

Varying



Functional Abstraction 2/3

```
# Draw a traffic light
above(circle(40, "solid", "red"),
  above(circle(40, "solid", "yellow"),
    circle(40, "solid", "green")))

# Can be changed to
fun bulb(color):
  circle(40, "solid", color)
end

above(bulb("red"),
  above(bulb("yellow"),
    bulb("green")))
```

Functional Abstraction: 3/3 (kicking it up a notch)

```
fun bulb(color):
  circle(40, "solid", color)
end

fun traffic-light():
  above(bulb("red"),
    above(bulb("yellow"),
      bulb("green")))
end
```

Functional Abstraction: Back To Baking

- Consider Mary's cake shop (again)
- We want to determine the price of each cake based on the cost of the ingredients and the time to prepare it.
- The price is twice the cost of the ingredients plus 1/4 of the preparation time in minutes.
- One approach: consider each cake, separately

Chocolate cake
Ingredients: $10
Preparation time: 20 minutes

```
choc-cake-price = (2 * 10) + (0.25 * 20)
```

Cheesecake
Ingredients: $15
Preparation time: 36 minutes

```
cheesecake-price = (2 * 15) + (0.25 * 36)
```

Functional Abstraction: Ace of ~~Cakes~~ Functions

- Looking more closely…
  - The price is twice the cost of the ingredients plus 1/4 of the preparation time in minutes.
- Use functions to avoid repetitive code when performing the same operations with different values.
- Purple font/arrows: different values? Hmm… should be a parameter!

Chocolate cake
Ingredients: $10
Preparation time: 20 minutes

```
choc-cake-price = (2 * 10) + (0.25 * 20)
```

Cheesecake
Ingredients: $15
Preparation time: 36 minutes

```
cheesecake-price = (2 * 15) + (0.25 * 36)
```

Cost of the ingredients

Preparation time

Informal Definition

- Parameters: generic names representing values that are passed into a function & are required/needed for creating a result.

```
fun
    cake-price(ingredients-cost, prep-time):
        (2 * ingredients-cost) + (0.25 * prep-time)
end
```

- Using our cake example, we can now calculate cost of any kind of cake by calling the cake-price function.

```
# Price of chocolate cake
 cake-price(10, 20)
# Price of cheesecake
 cake-price(15, 36)
```

Making ~~bitter batter better with butter~~ functions better

- Improved definition: Parameters: generic names representing values of a particular type that are passed into a function & are required/needed for creating a result.

```
fun
    cake-price(ingredients-cost :: Number, prep-time ::
Number):
        (2 * ingredients-cost) + (0.25 * prep-time)
end
```

- We specify the type of each parameter so that Pyret will check that the right type of values are actually being passed. Why does this matter?

Making ~~bitter batter better with butter~~ functions better

- Improved definition: Parameters: generic names representing values of a particular type that are passed into a function & are required/needed for creating a result.

```
fun
    cake-price(ingredients-cost :: Number, prep-time ::
Number):
        (2 * ingredients-cost) + (0.25 * prep-time)
end
```

- We specify the type of each parameter so that Pyret will check that the right type of values are actually being passed. Why does this matter?

  Answer: so that we can fail fast and discover problems faster than if we didn't check

Making functions better (returning a result)

- We can do this for the function result (i.e. return type) too!

```
fun
   cake-price(ingredients-cost :: Number, prep-time ::
Number) -> Number:
      (2 * ingredients-cost) + (0.25 * prep-time)
end
```

- We specify the type of the return value to maintain data consistency

  i.e. so that we can fail fast when calling a function and naming the result!

Using pyret


View
Connect to Google Drive
Run
Stop
use context essentials2021
fun
cake-price(ingredients-cost :: Number, prep-time :: Number) ->
Number:
(2 * ingredients-cost) + (0.25 * prep-time)
end
››› cake-price(5, 9)
12.25
››› cake-price("bam", "chocolate")
The Number annotation
definitions://:2:34-2:40
3 cake-price(ingredients-cost :: Number, prep-time :: Number) ->
Number:
was not satisfied by the value
"bam"
(Show program evaluation trace...)
››› cost = cake-price(5, 9)
›››

Making functions better (epilogue)

- It's a good idea to let ~~the instructor~~ other programmers know what a function does!

```
fun
   cake-price(ingredients-cost :: Number, prep-time ::
Number) -> Number:
doc: "Calculate price of cake based on ingredient cost and prep time"
      (2 * ingredients-cost) + (0.25 * prep-time)
end
```

- We document the function so that a user of the function, or one who maintains it, can understand and use it properly.

  i.e. so that we don't have to fail at all!

More On Failing Fast (i.e. testing)

- Consider the following function which includes some test information

```
fun cakes-to-make(num-sold :: Number) -> Number:
doc: "Compute the number of cakes to make based on
the previous number sold"
num-sold + 2
where:
   cakes-to-make(0) is 2
   cakes-to-make(107) is 109
End
```

- But what if we happens to make a typo in pyret…

More On Failing Fast (i.e. testing)

```
1  use context essentials2021
2  fun cakes-to-make(num-sold :: Number) -> Number: doc: "Compute the
   number of cakes to make based on the previous number sold"
3    num-sold + 22
4  where:
5     cakes-to-make(0) is 2
6     cakes-to-make(107) is 109
7  end
8
```

0	2
TESTS PASSED	TESTS FAILED

cakes-to-make — Hide Details

0 out of 2 tests passed in this block.

Test 1: Failed

The test operator `is` failed for the test:

definitions://:4:3-4:24

```
5    cakes-to-make(0) is 2
```

It succeeds only if the left side and right side are equal.

The left side was:

22

The right side was:

2

Test 2: Failed

The test operator `is` failed for the test:

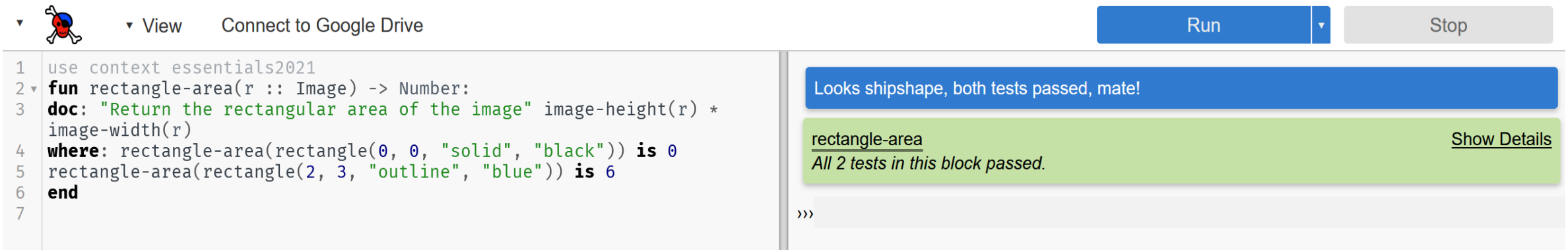
definitions://:5:3-5:28

Another Testing Example

- Consider the following function which uses an image!

```
fun rectangle-area(r :: Image) -> Number:
doc: "Return the rectangular area of the image"
image-height(r) * image-width(r)
where: rectangle-area(rectangle(0, 0, "solid",
"black")) is 0
rectangle-area(rectangle(2, 3, "outline", "blue"))
is 6
end
```

Another Testing Example: epilogue



Another Testing Example: epilogue