Problem Solving and Abstraction (CMPU 101)

Tom Ellman

Lecture 2

Comments in Pyret Code

- Text in the program file that Pyret should ignore.

- Line Comments: Ignore the rest of the line.

```
#cat = "Felix"
#CAT
```

- Block Comments: Ignore multiple lines.

```
#|
x = 17
y = 3
z = x + y
z
|#
```

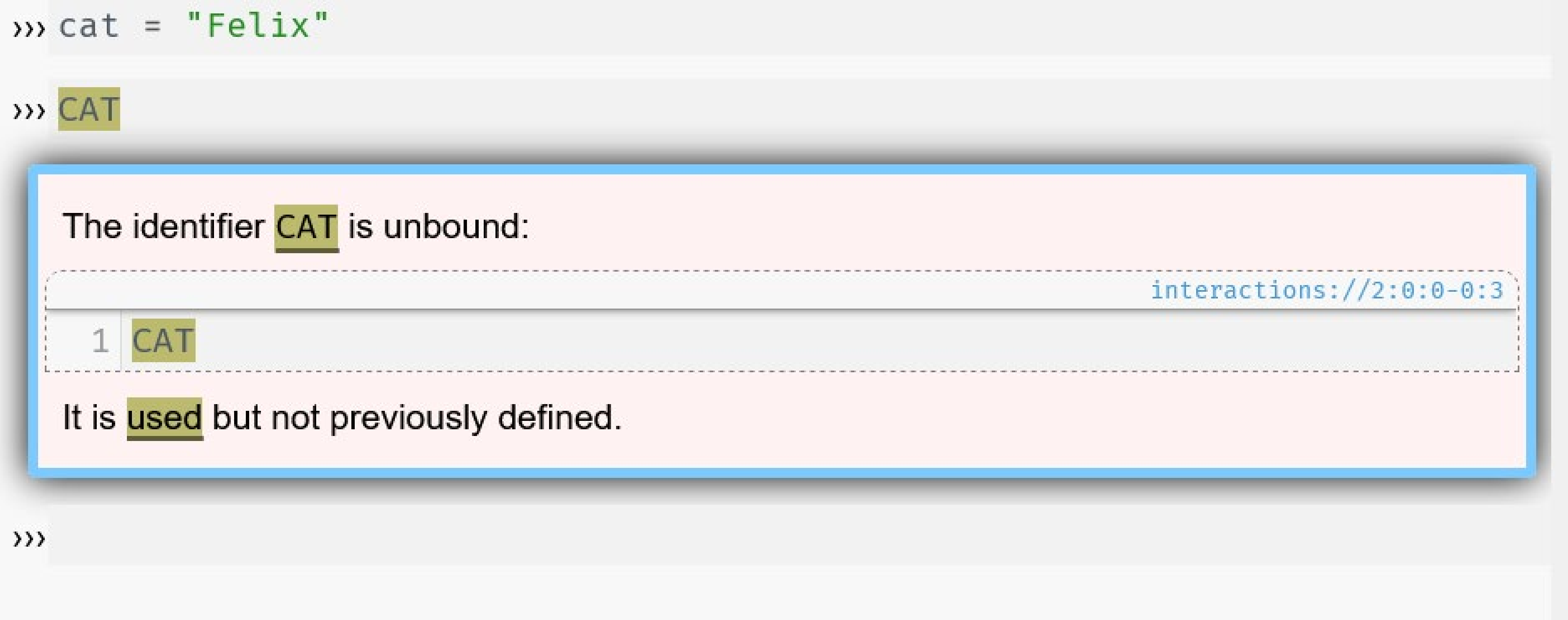
Names (a.k.a. Identifiers)

- A sequence of characters:
  - Letters: a … z or A … Z
  - Numerals 0 … 9
  - Punctuation: - (dash) or _ (underscore)
- Starting with a letter.

OK: x, y1, CAT, dog, pet-store, dog_food

Not OK: 137student, s!, t: , u*, $

Case Matters



```
››› cat = "Felix"

››› CAT
```

The identifier CAT is unbound:

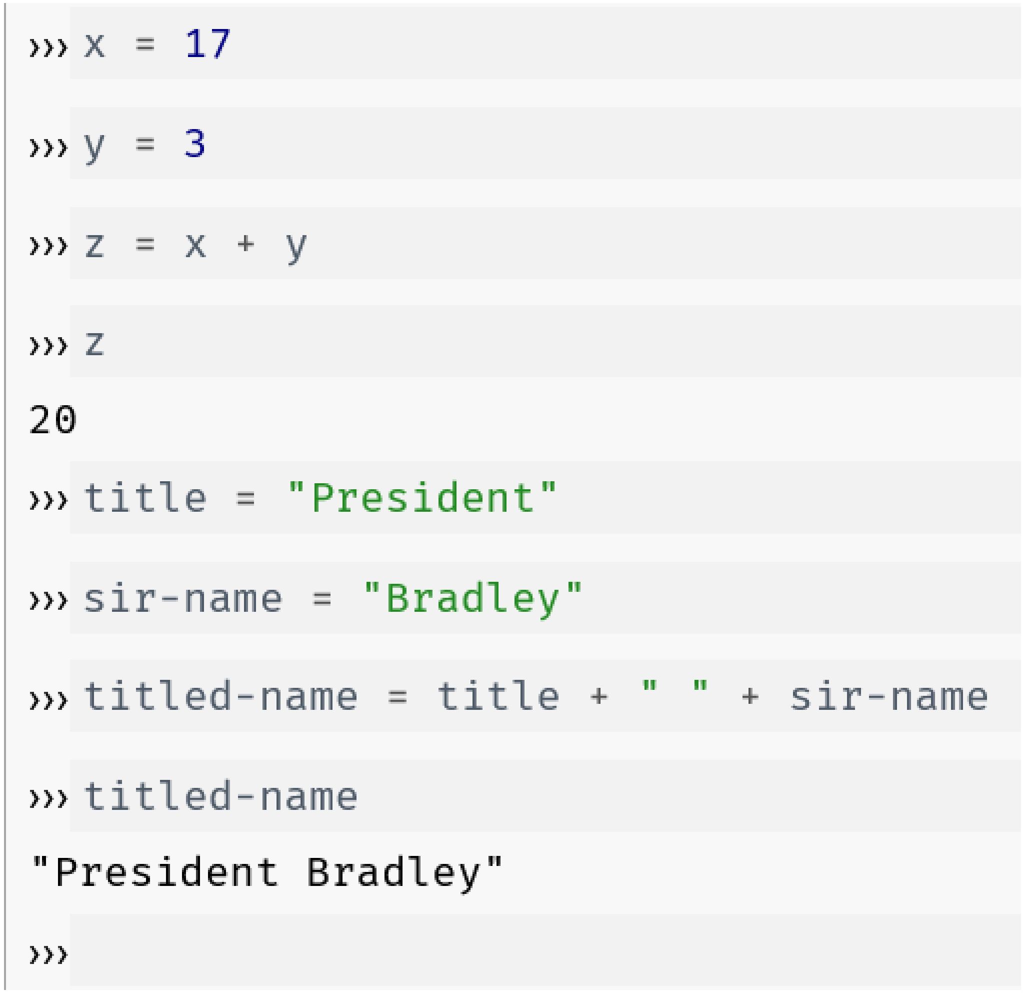
```
interactions://2:0:0-0:3
CAT
```

It is used but not previously defined.

```
›››
```

CAT and cat are different names.

Names



```
››› x = 17
››› y = 3
››› z = x + y
››› z
20
››› title = "President"
››› sir-name = "Bradley"
››› titled-name = title + " " + sir-name
››› titled-name
"President Bradley"
›››
```

Definitions

x = 17 and **title = "President"** are definitions.

A definition creates a binding that associates a name with a value by storing them in Pyret's internal directory.

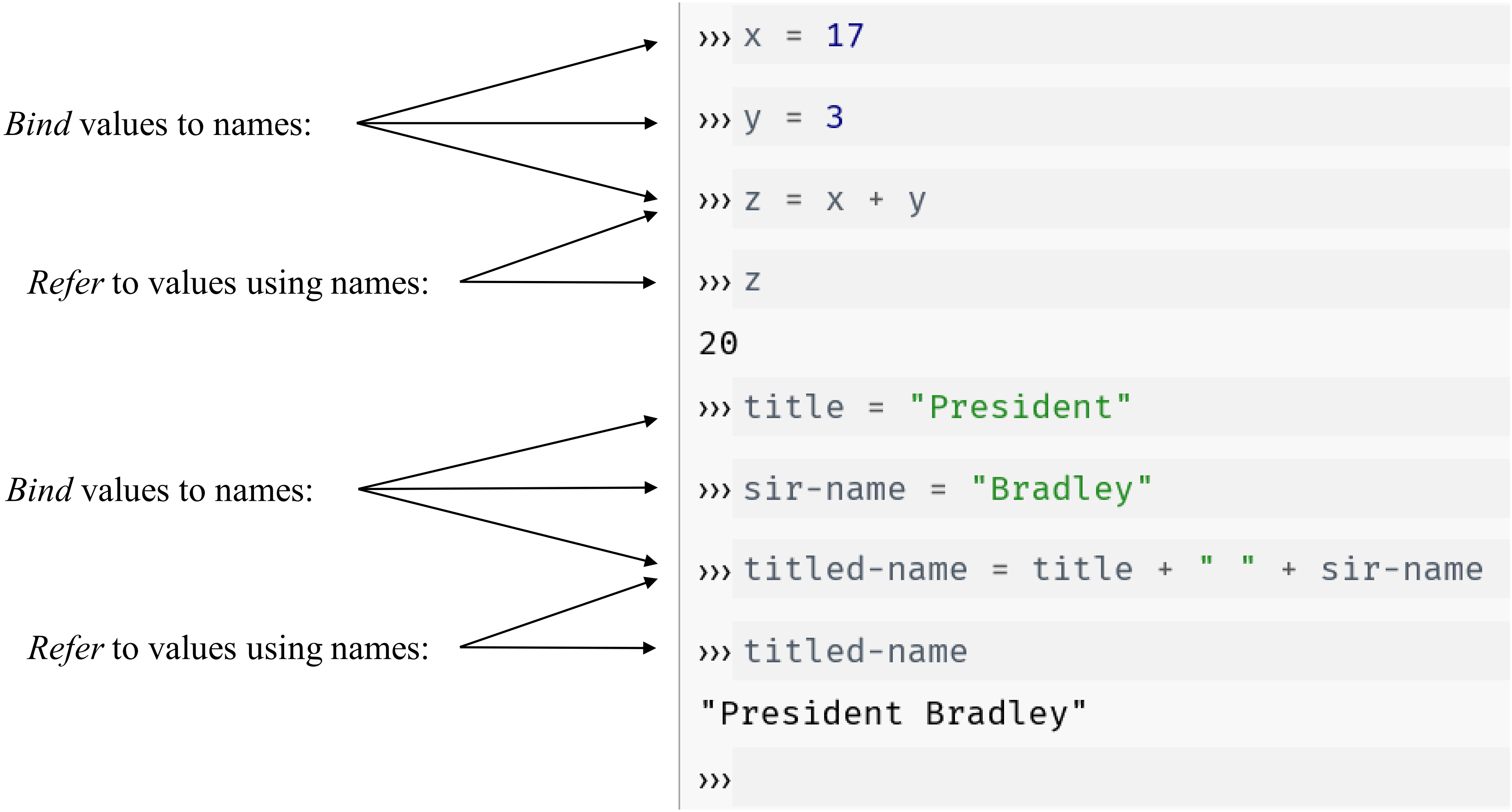
If Pyret comes across the name later, Pyret will replace the name with it's associated value.

In Pyret, definitions are sometimes called declarations. In other languages, e.g., Java, C++ these are different concepts.

Definitions versus Expressions

- Notice that Pyret does not display a value after processing a definition.
- Definitions are statements, not expressions.
- Statements are not evaluated to produce values.
- Instead, statements cause side effects of changing Pyret’s directory .

Binding and Reference



Bind values to names:

Refer to values using names:

Bind values to names:

Refer to values using names:

```
››› x = 17

››› y = 3

››› z = x + y

››› z
20
››› title = "President"

››› sir-name = "Bradley"

››› titled-name = title + " " + sir-name

››› titled-name
"President Bradley"
›››
```

Pyret’s Internal Directory

Name	**Value**
x	17
y	3
z	20
title	“President”
sir-name	“Bradley”
titled-name	“President Bradley”

Notice that the directory stores values, not expressions:

the value **20** rather than the expression **x + y**

and

the value **“President Bradley”** rather than the expression **title + " " + sir-name**.

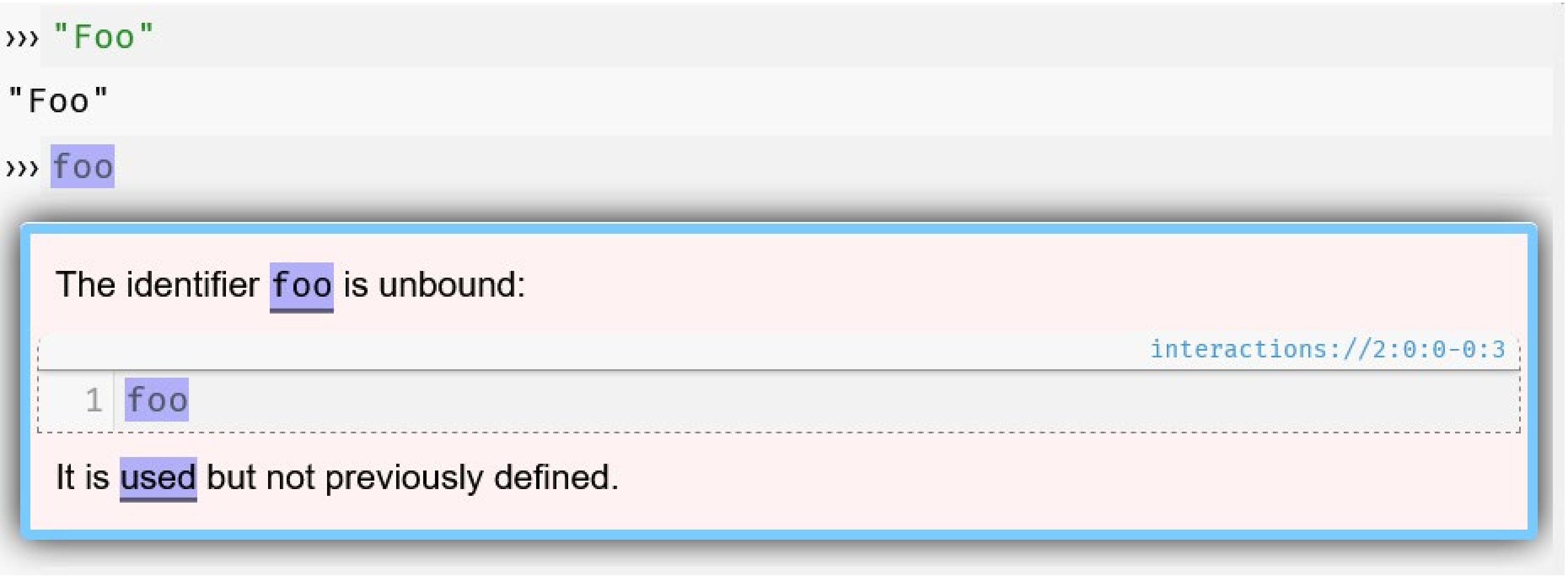
Substitution

When Pyret evaluates the expression:

x + y

it finds the values of **x** and **y** in the directory and

replaces **x** and **y** with their respective values to get: **17 + 3** which evaluated to **20**



```
››› "Foo"
"Foo"
››› foo
```

The identifier foo is unbound:

```
interactions://2:0:0-0:3
1  foo
```

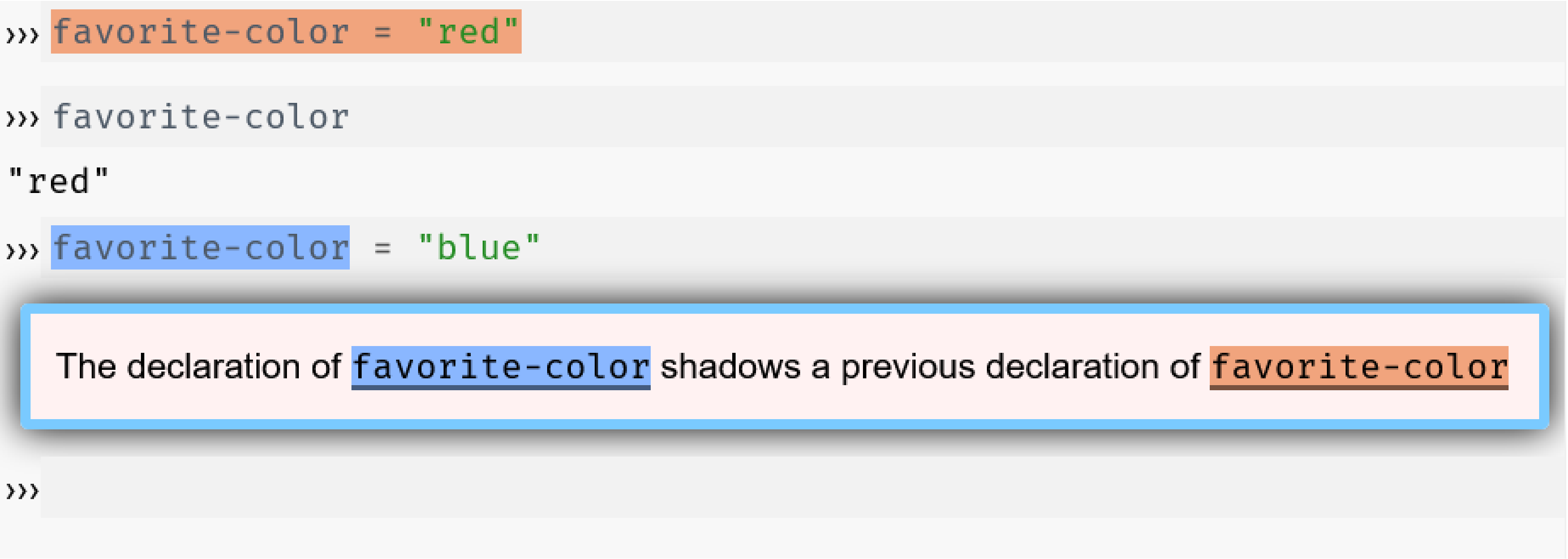
It is used but not previously defined.

“foo” is a string.

foo (without quotes) is potentially a name.

Since **foo** has not been defined, Pyret says it is unbound.

An error occurs if we try to use **foo** as a name before we have given it a definition.



```
››› favorite-color = "red"
››› favorite-color
"red"
››› favorite-color = "blue"
```

The declaration of `favorite-color` shadows a previous declaration of `favorite-color`

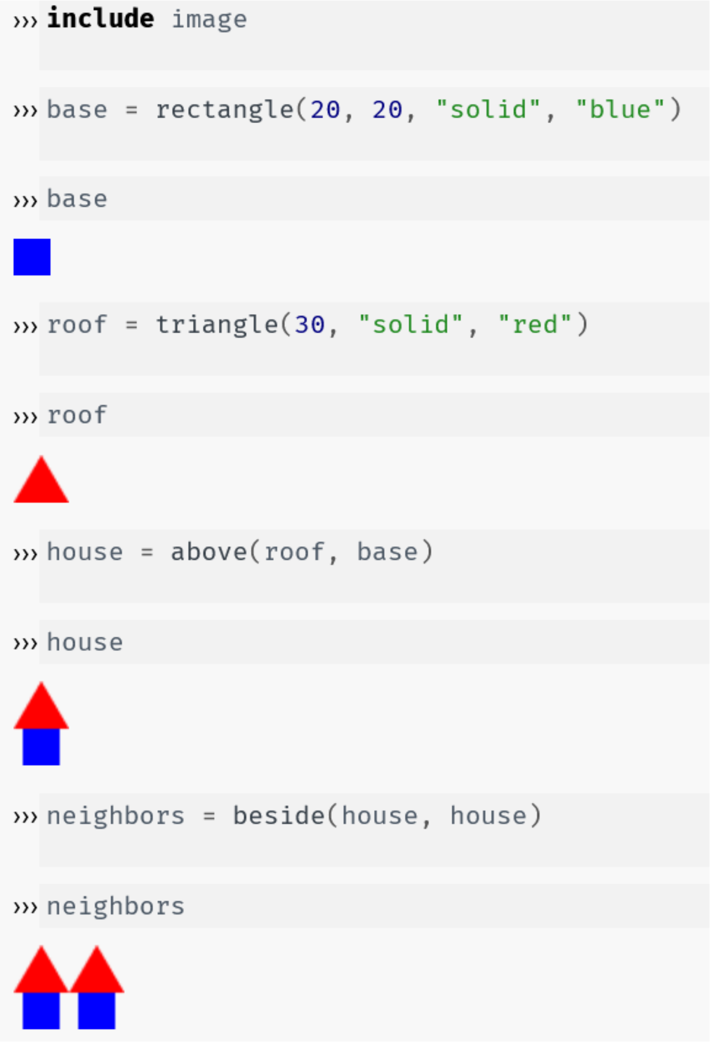
```
›››
```

Once we define a name, we (normally) cannot change its value. If we try to do so, we get an shadow error.

Try These Expressions!

```
include image
base = rectangle(20, 20, "solid", "blue")
base
roof = triangle(30, "solid", "red")
roof
house = above(roof, base)
house
neighbors = beside(house, house)
neighbors
```

Building Expressions from Sub-Expressions



```
››› include image

››› base = rectangle(20, 20, "solid", "blue")

››› base

››› roof = triangle(30, "solid", "red")

››› roof

››› house = above(roof, base)

››› house

››› neighbors = beside(house, house)

››› neighbors
```

Sub-Sub-Expression

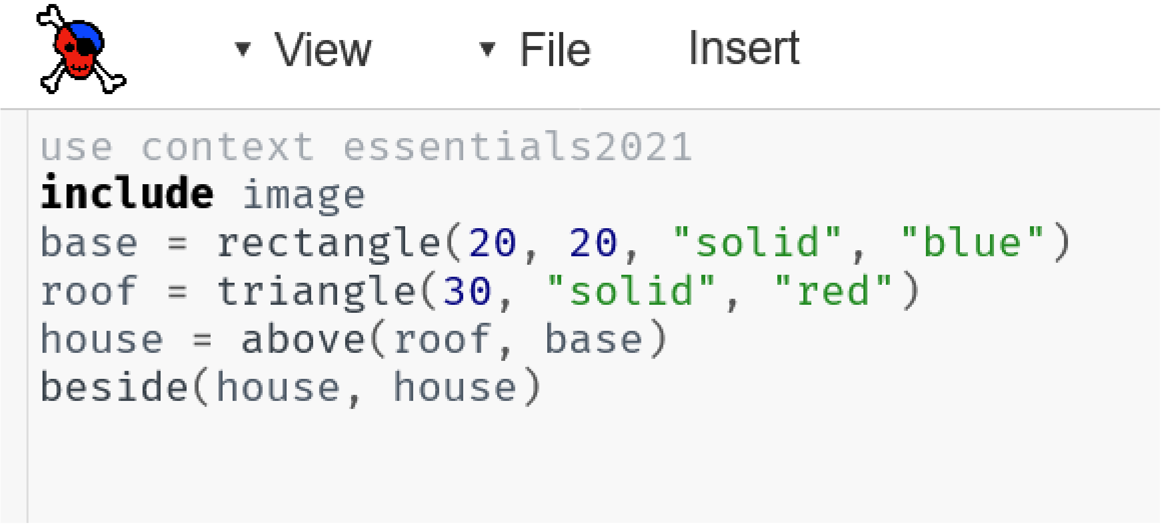
Sub-Sub-Expression

Sub-Expression

Expression

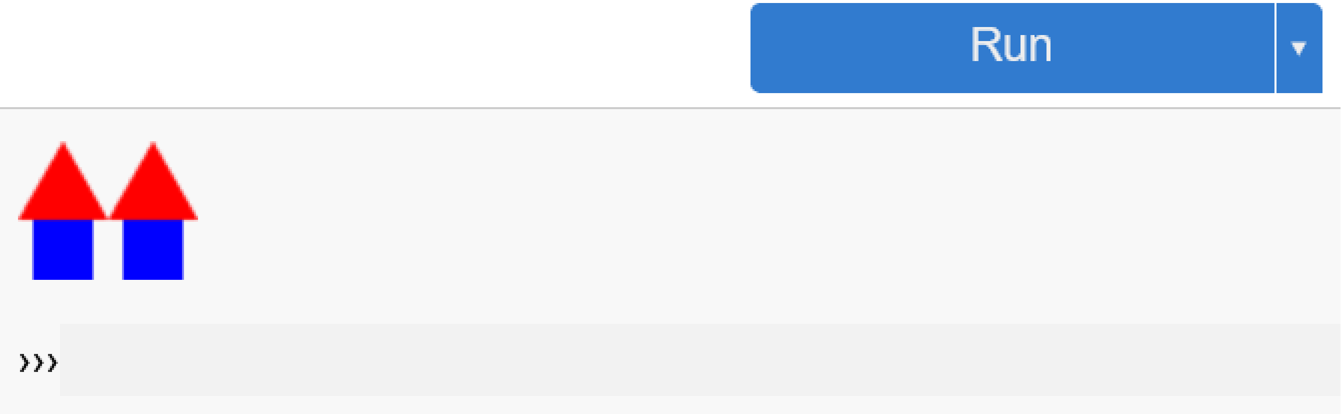
Cut and Paste into the Definitions Pane & Press **Run**

```
include image
base = rectangle(20, 20, "solid", "blue")
roof = triangle(30, "solid", "red")
house = above(roof, base)
beside(house, house)
```



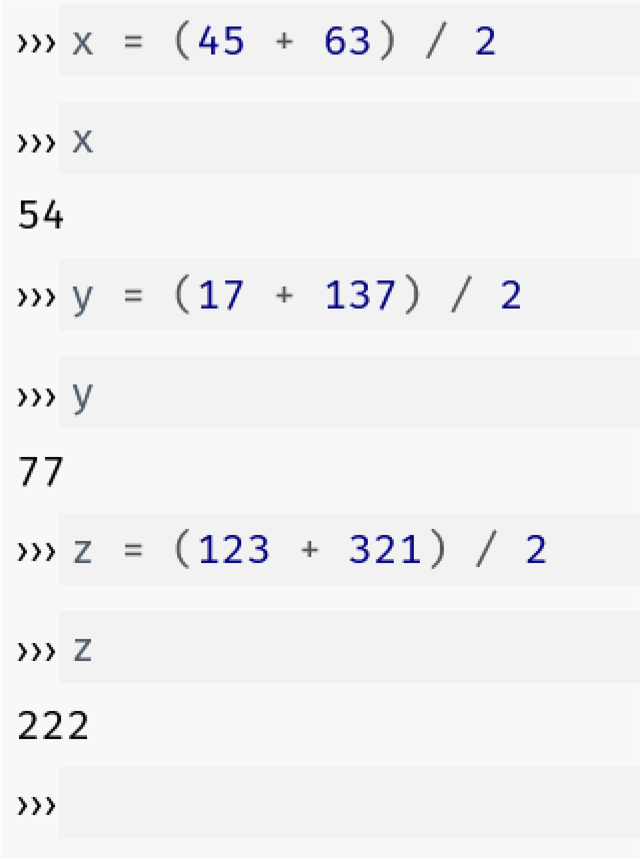
```
use context essentials2021
include image
base = rectangle(20, 20, "solid", "blue")
roof = triangle(30, "solid", "red")
house = above(roof, base)
beside(house, house)
```

Definitions



Interactions

Repeated Similar Expressions



```
>>> x = (45 + 63) / 2
>>> x
54
>>> y = (17 + 137) / 2
>>> y
77
>>> z = (123 + 321) / 2
>>> z
222
>>>
```

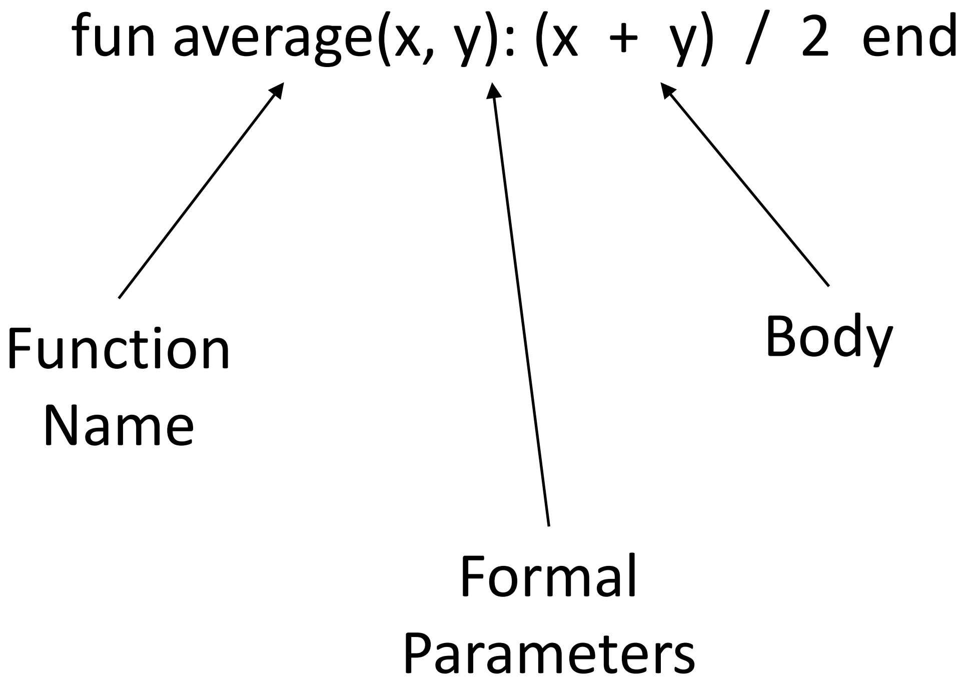
We are typing and evaluating similar expressions over and over. This could get tedious.

In each case, we compute a sum and divide by two.

We can capture this pattern in a function definition.

DRY Principle: Don't Repeat Yourself.

Defining a Function



```
fun average(x, y): (x  +  y)  /  2  end
```

Function

Name

Body

Formal

Parameters

Function definition creates a new binding in the Pyret’s directory between the function name and its definition. The body is not evaluated at this time.

Functional Abstraction

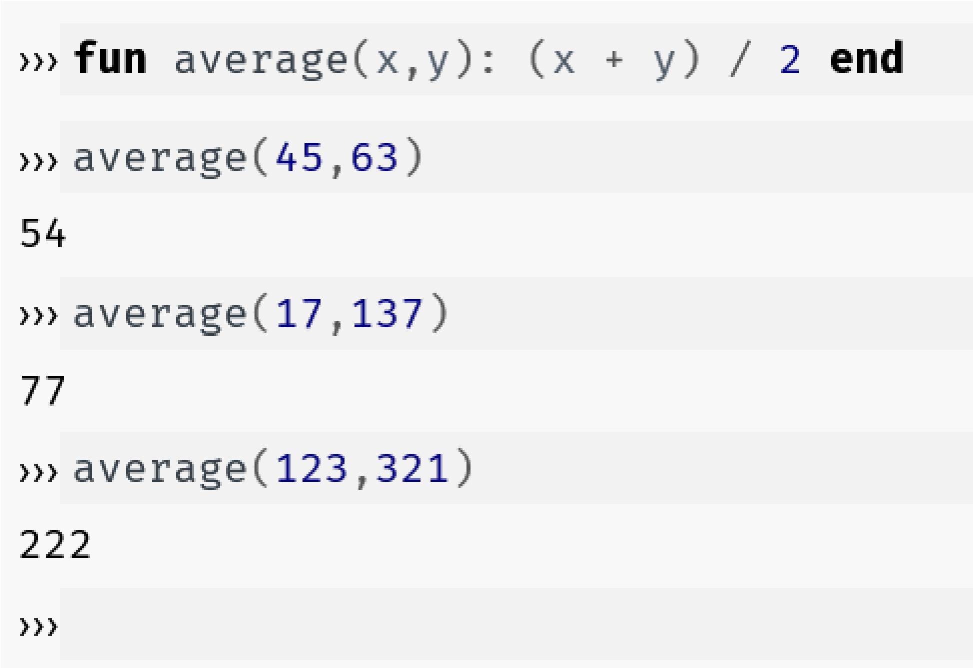
$$\left.\begin{array}{l}(45 + 53) / 2 \\ \\ (17 + 137) / 2 \\ \\ (123 + 321) / 2\end{array}\right\} \quad (x + y) / 2$$

We replace the common parts of these expressions by new names, called formal parameters.

Function Definition Format

```
fun <function name>(<parameters>):
  <body expression>
end
```

```
fun average(x,y):
   (x  +  y)  /  2
end
```

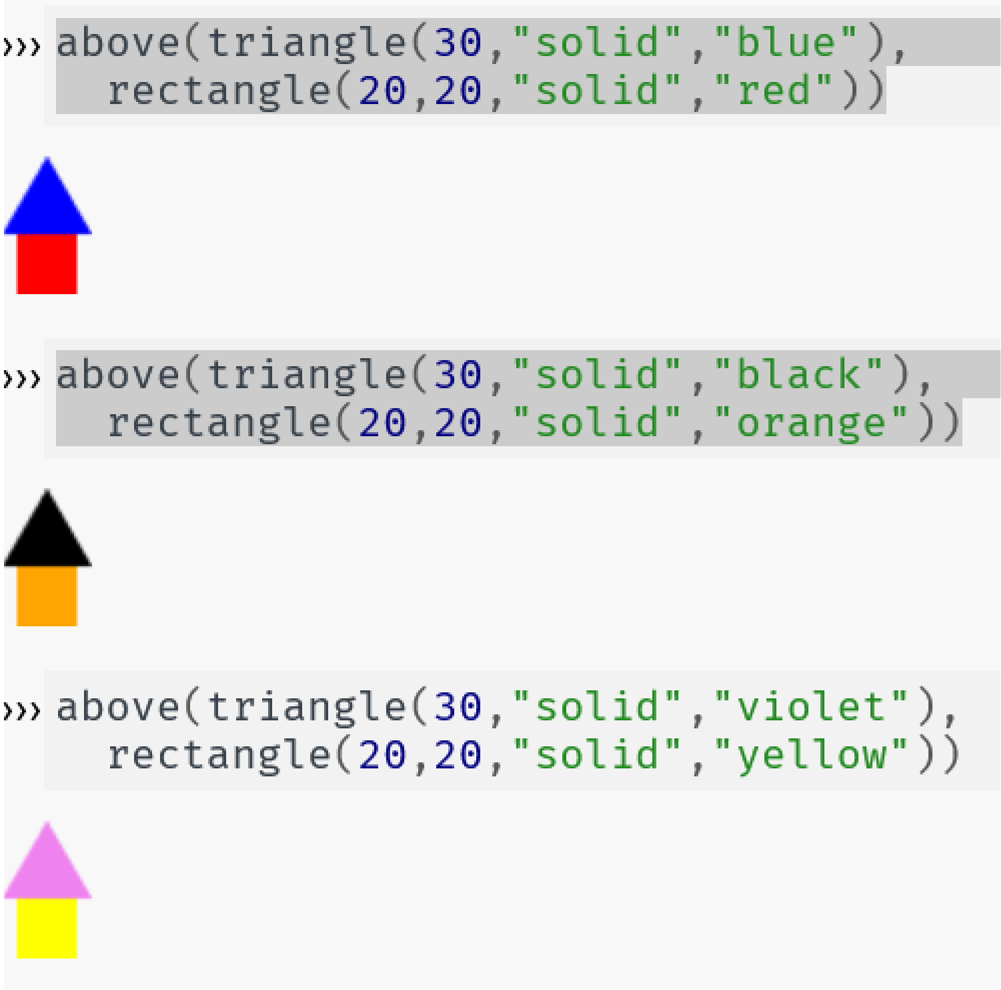


```
››› fun average(x,y): (x + y) / 2 end
››› average(45,63)
54
››› average(17,137)
77
››› average(123,321)
222
›››
```

Pyret Evaluating: **average(45,63)**

1. Find the definition of **average** in the directory.
2. Associate the formal parameters **(x,y)** in the definition with the actual parameters **(45,63)** and create temporary bindings: **x = 45** and **y = 63**.
3. Evaluate the body expression **(x + y) / 2** using the new bindings for **x** and **y** to get the value **54** of **average(45,63)**.

Repeated Similar Expressions



```
››› above(triangle(30,"solid","blue"),
      rectangle(20,20,"solid","red"))
```

```
››› above(triangle(30,"solid","black"),
      rectangle(20,20,"solid","orange"))
```

```
››› above(triangle(30,"solid","violet"),
      rectangle(20,20,"solid","yellow"))
```

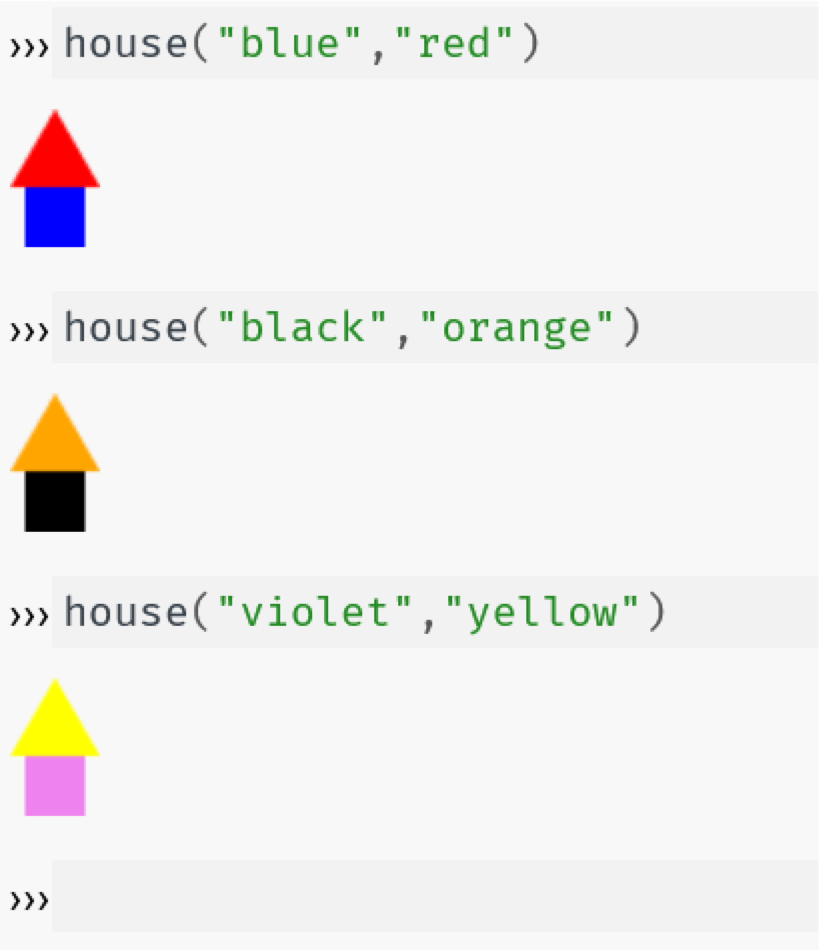
We are typing and evaluating similar expressions over and over. This could get tedious.

How are the expressions similar? How are they different?

Can can capture this pattern in a function definition?

DRY Principle: Don't Repeat Yourself.

```
fun house(base-color,roof-color):
 above(triangle(30,"solid",roof-color),
  rectangle(20,20,"solid",base-color))
end
```



```
››› house("blue","red")

››› house("black","orange")

››› house("violet","yellow")

›››
```

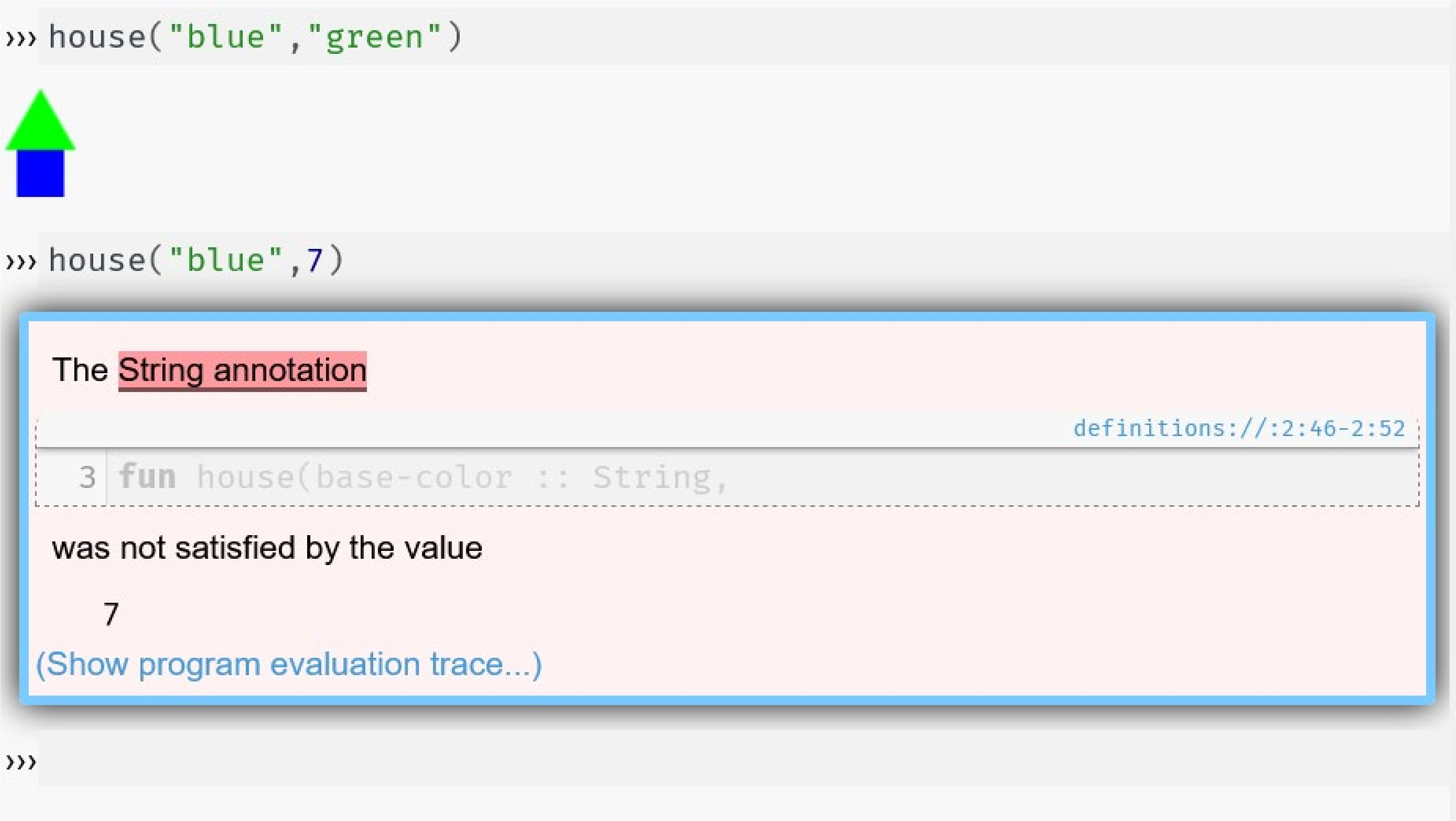
Annotations and Contracts

```
fun house(base-color :: String,
                    roof-color :: String) -> Image:
  above(triangle(30,"solid",roof-color),
    rectangle(20,20,"solid",base-color))
end
```

We use **:: String** to require the user of **house** to provide only data of type **String** for **base-colo**r and **roof-color**.

We use **-> Image** to promise to the user that the **house** function will return a datum of type **Image**.

The requirement on the user of the house function and the promise made by the programmer are a **contract**. If both full the terms of the contract there will not be any data-type errors.



```
››› house("blue","green")

››› house("blue",7)

The String annotation

                                                        definitions://:2:46-2:52
fun house(base-color :: String,

was not satisfied by the value

   7
(Show program evaluation trace...)

›››
```

Pyret detects the data-type error while evaluating the expression: **house(“blue”,7)**.

Solving the Quadratic Equation

$a x^2 + b x + c = 0$	Solve for x
quad-high	High Root
quad-low	Low Root

High Root

```
fun quad-high(a :: Number,
              b :: Number,
              c :: Number) -> Number:
((0 - b) + num-sqrt((b * b) - (4 * (a * c)))) / (2 * a)
end
```

Test Cases

```
fun quad-high(a :: Number,
              b :: Number,
              c :: Number) -> Number:
((0 - b) + num-sqrt((b * b) - (4 * (a * c)))) / (2 * a)
where:
    quad-high(1,0,-4) is 2
    quad-high(1,-5,4) is 4
end
```

Low Root

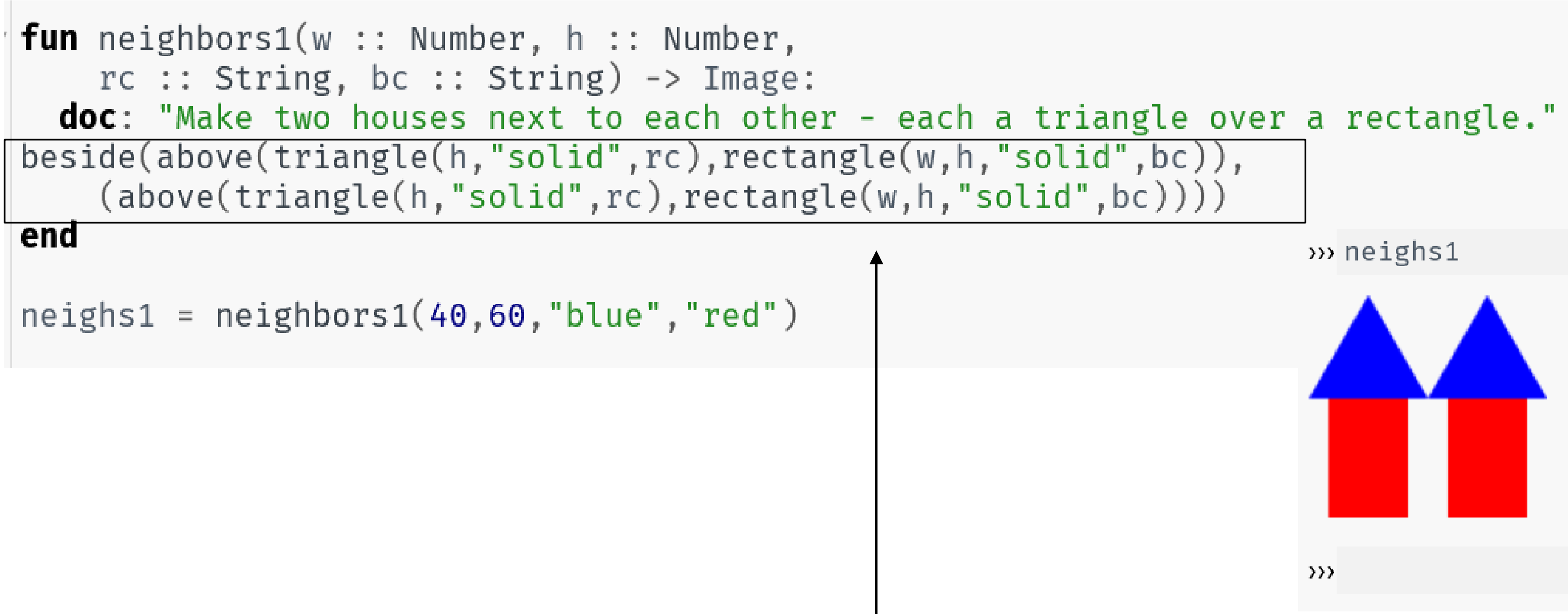
```
fun quad-low(a :: Number,
             b :: Number,
             c :: Number) -> Number:
((0 - b) - num-sqrt((b * b) - (4 * (a * c)))) / (2 * a)
end
```

Test Cases

```
fun quad-low(a :: Number,
             b :: Number,
             c :: Number) -> Number:
((0 - b) - num-sqrt((b * b) - (4 * (a * c)))) / (2 * a)
where:
    quad-low(1,0,-4) is -2
    quad-low(1,-5,4) is 1
end
```

Helper Functions and (DRS)

- Avoid repeated typing.
- Avoid repeated evaluation.
- Make code more readable.



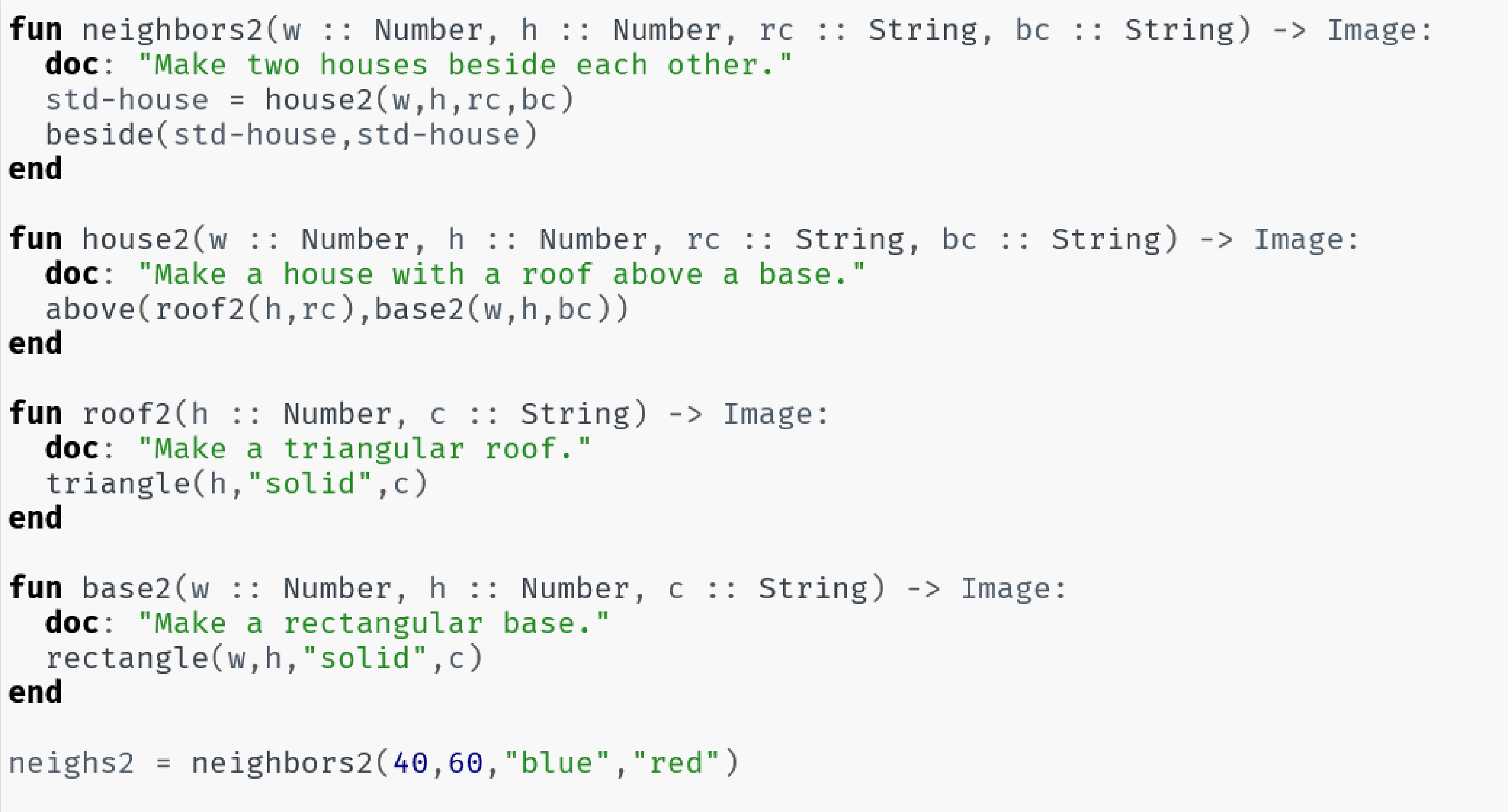
```
fun neighbors1(w :: Number, h :: Number,
    rc :: String, bc :: String) -> Image:
  doc: "Make two houses next to each other - each a triangle over a rectangle."
beside(above(triangle(h,"solid",rc),rectangle(w,h,"solid",bc)),
    (above(triangle(h,"solid",rc),rectangle(w,h,"solid",bc))))
end

neighs1 = neighbors1(40,60,"blue","red")
```

```
››› neighs1

›››
```

We typed some expressions twice.

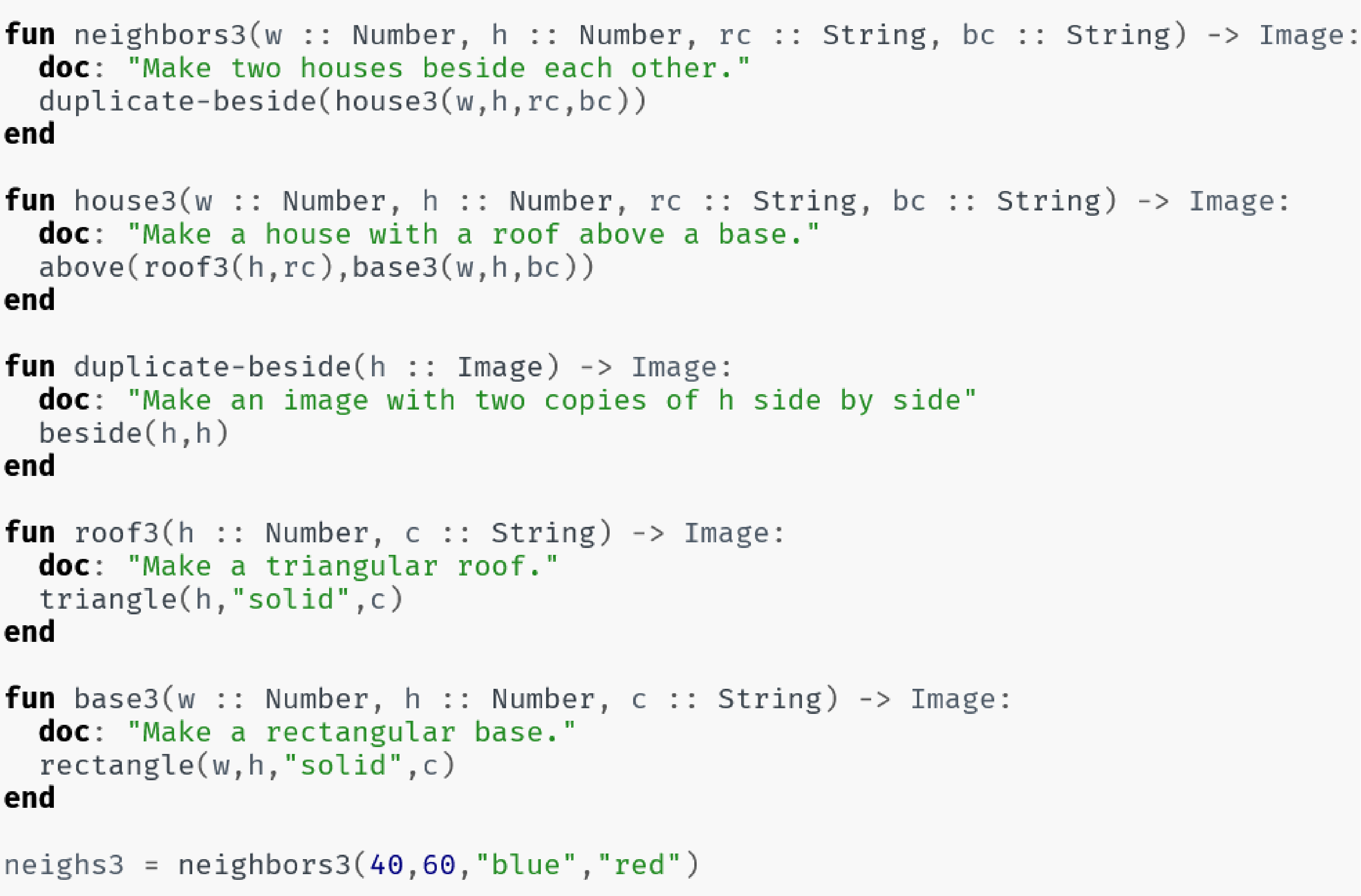
We are evaluating some expressions twice.

This is not so easy to read and understand.



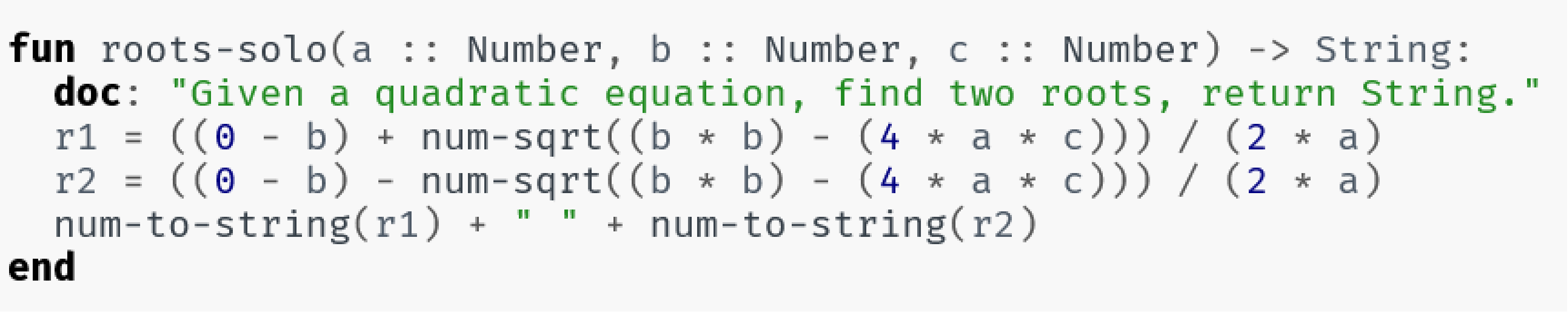
```
fun neighbors2(w :: Number, h :: Number, rc :: String, bc :: String) -> Image:
  doc: "Make two houses beside each other."
  std-house = house2(w,h,rc,bc)
  beside(std-house,std-house)
end

fun house2(w :: Number, h :: Number, rc :: String, bc :: String) -> Image:
  doc: "Make a house with a roof above a base."
  above(roof2(h,rc),base2(w,h,bc))
end

fun roof2(h :: Number, c :: String) -> Image:
  doc: "Make a triangular roof."
  triangle(h,"solid",c)
end

fun base2(w :: Number, h :: Number, c :: String) -> Image:
  doc: "Make a rectangular base."
  rectangle(w,h,"solid",c)
end

neighs2 = neighbors2(40,60,"blue","red")
```

Our use of helper functions (house, roof, base) make the program easier to read and understand. But we still type and evaluate an expression twice. How can we fix this problem?



```
fun neighbors3(w :: Number, h :: Number, rc :: String, bc :: String) -> Image:
  doc: "Make two houses beside each other."
  duplicate-beside(house3(w,h,rc,bc))
end

fun house3(w :: Number, h :: Number, rc :: String, bc :: String) -> Image:
  doc: "Make a house with a roof above a base."
  above(roof3(h,rc),base3(w,h,bc))
end

fun duplicate-beside(h :: Image) -> Image:
  doc: "Make an image with two copies of h side by side"
  beside(h,h)
end

fun roof3(h :: Number, c :: String) -> Image:
  doc: "Make a triangular roof."
  triangle(h,"solid",c)
end

fun base3(w :: Number, h :: Number, c :: String) -> Image:
  doc: "Make a rectangular base."
  rectangle(w,h,"solid",c)
end

neighs3 = neighbors3(40,60,"blue","red")
```

Using one more helper function, we have avoided typing (or evaluating) the same expression twice.

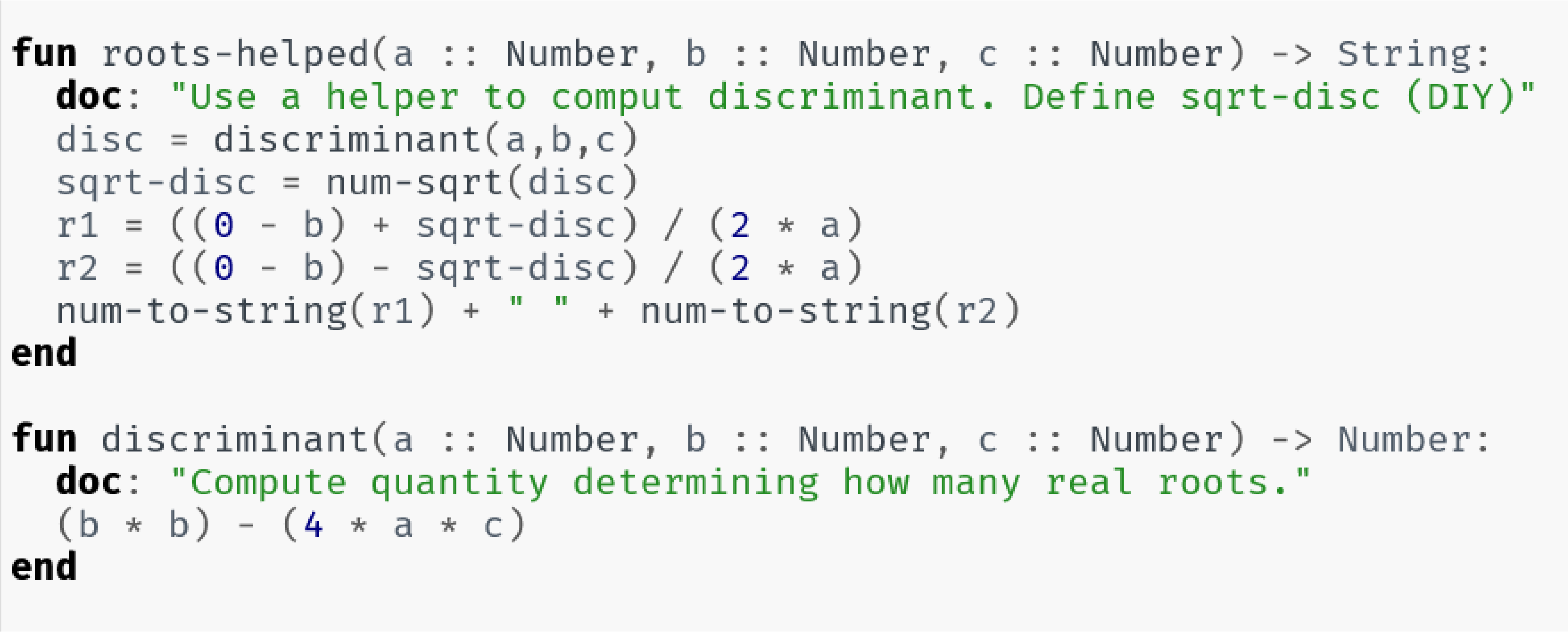


```
fun roots-solo(a :: Number, b :: Number, c :: Number) -> String:
  doc: "Given a quadratic equation, find two roots, return String."
  r1 = ((0 - b) + num-sqrt((b * b) - (4 * a * c))) / (2 * a)
  r2 = ((0 - b) - num-sqrt((b * b) - (4 * a * c))) / (2 * a)
  num-to-string(r1) + " " + num-to-string(r2)
end
```

We typed some expressions twice.

We are evaluating some expressions twice.

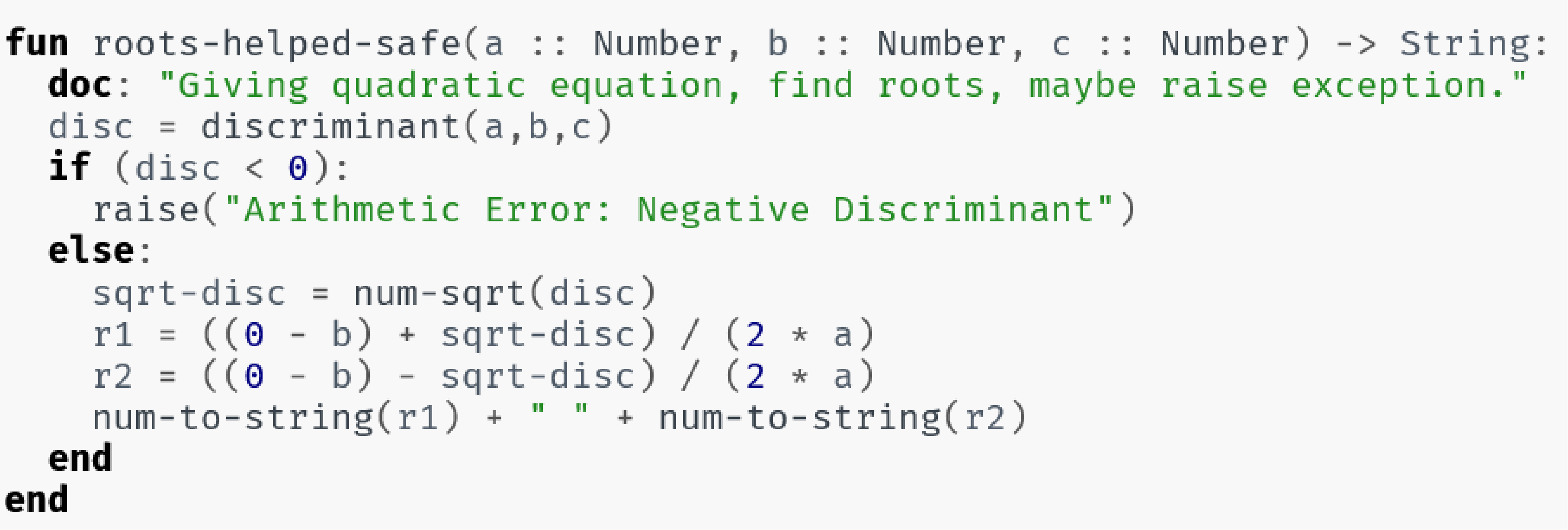
How can we fix these problems?



```
fun roots-helped(a :: Number, b :: Number, c :: Number) -> String:
  doc: "Use a helper to comput discriminant. Define sqrt-disc (DIY)"
  disc = discriminant(a,b,c)
  sqrt-disc = num-sqrt(disc)
  r1 = ((0 - b) + sqrt-disc) / (2 * a)
  r2 = ((0 - b) - sqrt-disc) / (2 * a)
  num-to-string(r1) + " " + num-to-string(r2)
end

fun discriminant(a :: Number, b :: Number, c :: Number) -> Number:
  doc: "Compute quantity determining how many real roots."
  (b * b) - (4 * a * c)
end
```

Now we are not computing the discriminant twice nor computing its square root twice.

Why is it tricky to eliminate the remaining duplication?



```
fun roots-helped-safe(a :: Number, b :: Number, c :: Number) -> String:
  doc: "Giving quadratic equation, find roots, maybe raise exception."
  disc = discriminant(a,b,c)
  if (disc < 0):
    raise("Arithmetic Error: Negative Discriminant")
  else:
    sqrt-disc = num-sqrt(disc)
    r1 = ((0 - b) + sqrt-disc) / (2 * a)
    r2 = ((0 - b) - sqrt-disc) / (2 * a)
    num-to-string(r1) + "  " + num-to-string(r2)
  end
end
```

Here we are solving a different problem. What happens if one or both roots are complex (not real) numbers?

We check whether the discriminant is negative. If so, we raise an exception. The `raise` function halts evaluation and displays its parameter.