



Expressions, Values, & Names

CMPU 101 – Problem Solving and Abstraction

Peter Lemieszewski



Programs

- A program (or script) instructs a computer to do something.
- These instructions must be very specific for the computer to carry them out.
 - Recall my National Engineers week comments
- But programs also need to be understood by people, i.e. they must be readable!

More Basics



- To write a program, we need to use a programming language
 - We could write the 1's and 0's (apologies to Elle King) in a way that the computer can understand the input stream... that's what assembly language is for, btw
- and programming environment. Also known as an Interactive Development Environment, IDE
- We write our computation in the (specified) programming language.
- We run the program in the environment.
 - There's more to the story, but this will suffice for now.

Introducing our IDE



- ...Both sides now

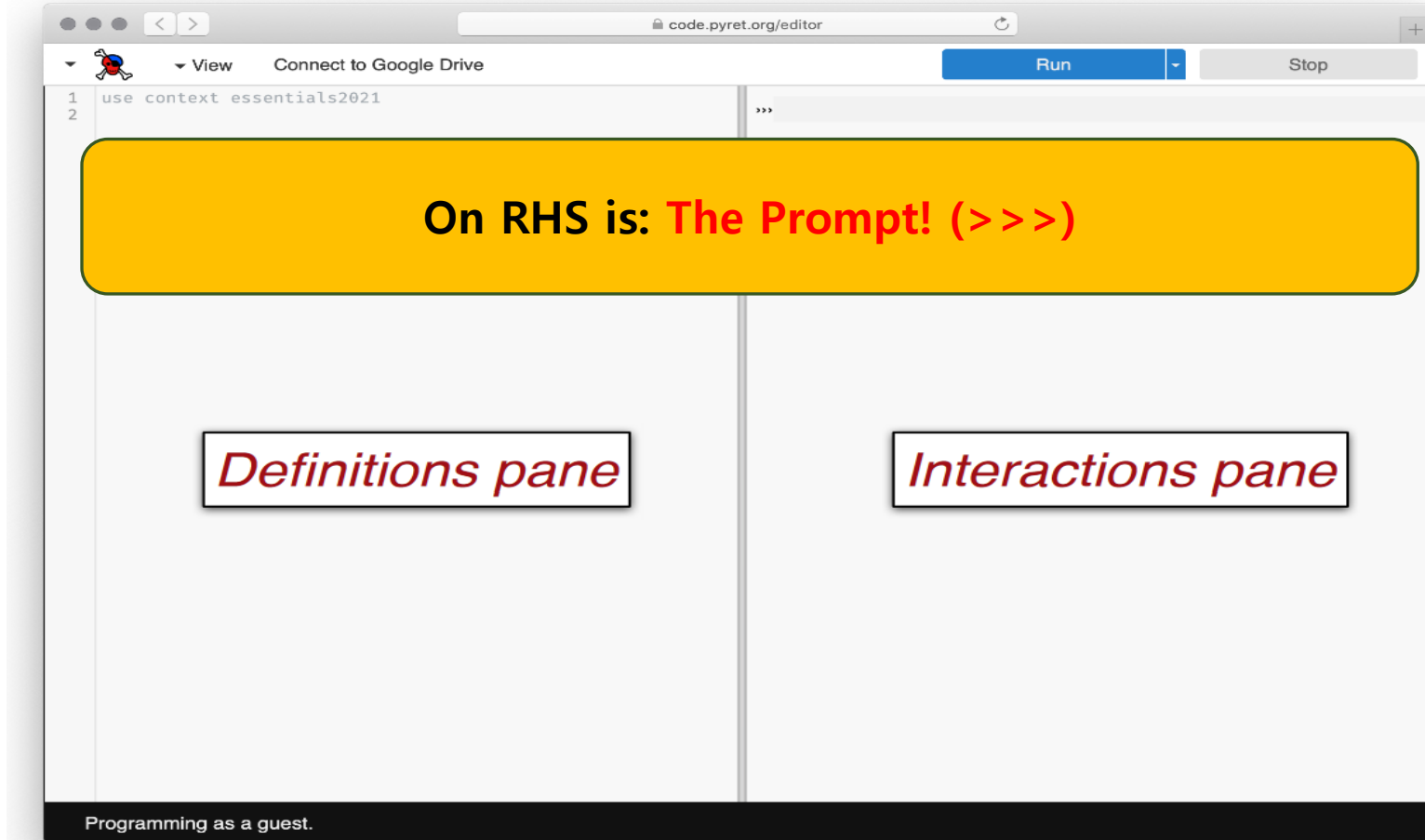


code.pyret.org

Introducing our IDE



- From up and down



code.pyret.org



Things you can do LHS/RHS

- And still somehow

The screenshot shows the code.pyret.org editor interface. The left pane is labeled "Definitions pane" and contains a yellow box with the following list:

- Write expressions
- Name expressions
- Use previously defined expressions

Below this box is a white box with the text *Definitions pane*. At the bottom of the left pane is another yellow box with the list:

- Save your code!

The right pane is labeled "Interactions pane" and contains a yellow box with the following list:

- Try out expressions
- Check syntax

Below this box is a white box with the text *Interactions pane*. The editor interface also shows a code editor with the text "use context essentials2021" and a "Run" button.

code.pyret.org

Pop Quiz!



Which pane would I use if...

1. I want to see if I can make a blue circle?
2. I want to define `my-shape` as a blue circle and use it later in my code?
3. I want to see if Pyret will accept this: `print "5"`?
4. I want to start my assignment now and finish it later?

Let's start to program by considering... Flags ?



Armenia



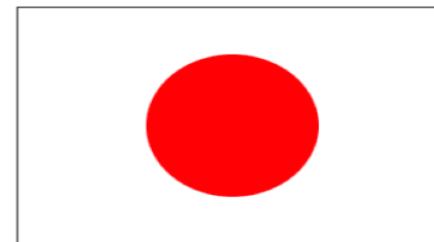
Austria



Colombia



Zambia

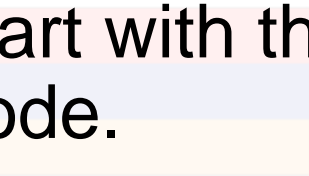


Japan

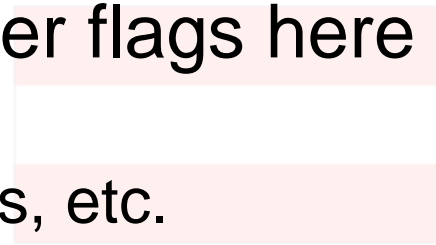


OK, we want to print some flags...

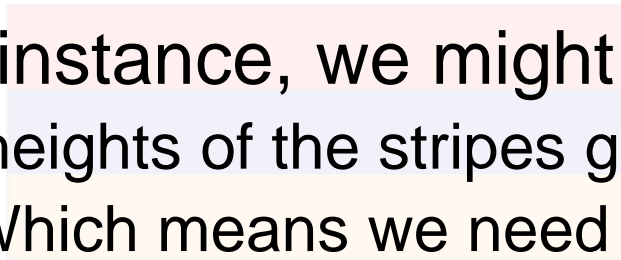
- Let's start with the data – consider flags here - before we dive in and write code.
 - Dimensions, shapes, juxtapositions, etc.
- For instance, we might want to compute
 - heights of the stripes given: overall flag dimensions,
 - Which means we need to write programs over [the set of] **numbers**.
- We also need a way to describe **colors** to our program.
- More generally, we need a way to create **images**
 - based on simple shapes of different colors.



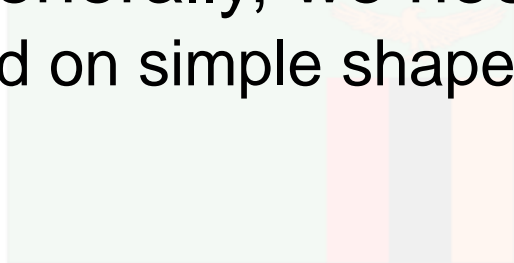
Armenia



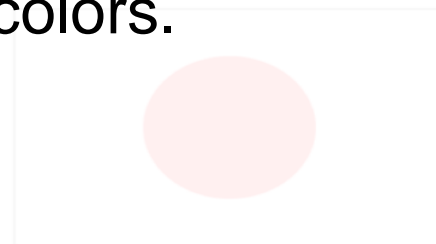
Austria



Colombia



Zambia



Japan

Numbers



- Consider
 - An individual number like 5 is a value – it can't be computed any further.
 - An expression like $(3 + 4) * (5 + 1)$ is a computation that produces an answer.
 - A program – any program - consists of one or more computations
 - Question: what about $3 + 4 * 5 + 1$?
 - WWJD?

Numbers



- Consider
 - An individual number like 5 is a value – it can't be computed any further.
 - An expression like $(3 + 4) * (5 + 1)$ is a computation that produces an answer.
 - A program – any program - consists of one or more computations
 - Question: what about $3 + 4 * 5 + 1$?
 - WWJD? See...
<https://introc.cs.princeton.edu/java/11precedence/>

In pyret...



```
>>> 3 + 4 * 5
```

Reading this expression errored:

[interactions://1:0:0-0:9](#)

```
1 | 3 + 4 * 5
```

The **+** and ***** operations are at the same grouping level. Add parentheses to group the operations, and make the order of operations clear.

```
>>> 3 + (4 * 5)
```

```
23
```

```
>>> |
```

Colors



- Consider
 - Names can be given as text strings, e.g.,
"purple"
 - Pyret will understand what "purple" means in the context of a color, i.e. if pyret is expecting a text string that represents a color. Let's clarify...

Shapes



- Consider

- ```
>>> include image
```
  - ```
>>> circle(50, "solid", "purple")
```

- We're asking pyret to create an image, specifically a solid purple circle with some dimension of 50.

Shapes



- Like numbers, we can manipulate images...
 - Numbers can be added, subtracted, etc.
 - Similarly, Images can overlaid, rotated, flipped, etc.

Moving On To Evaluations



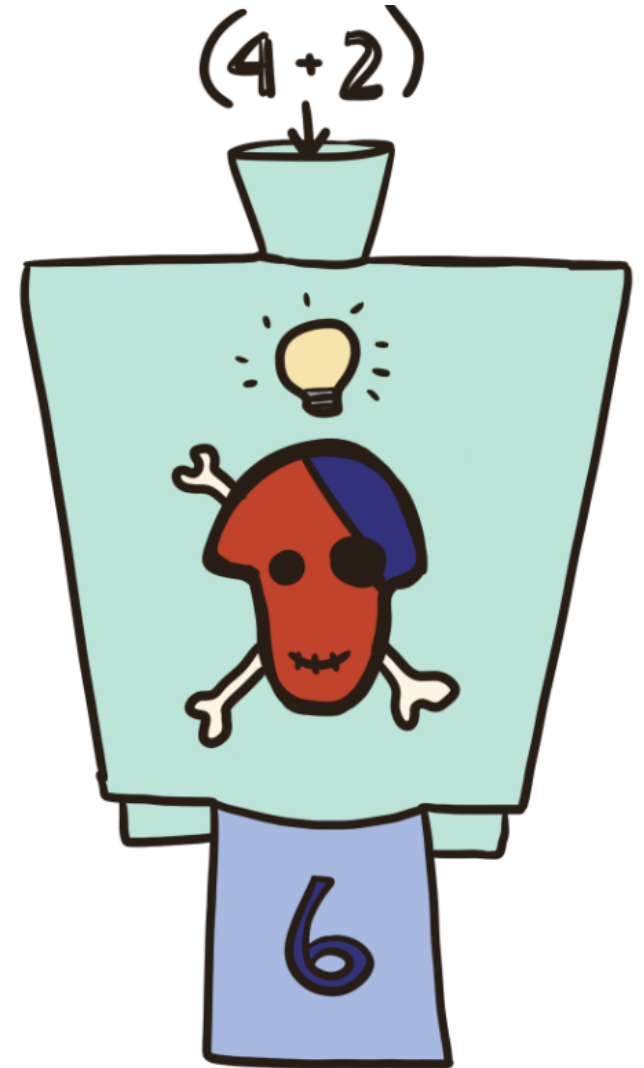
How does something like $(4 + 2) / 3$ work?

What is the operator $/$ dividing?

Shouldn't $/$ expect two numbers?

Even though $(4 + 2)$ isn't a number, it's an expression that *evaluates* to a number.

This works for all data types, not just numbers!



Moving On To Evaluations



How does something like $(4 + 2) / 3$ work?

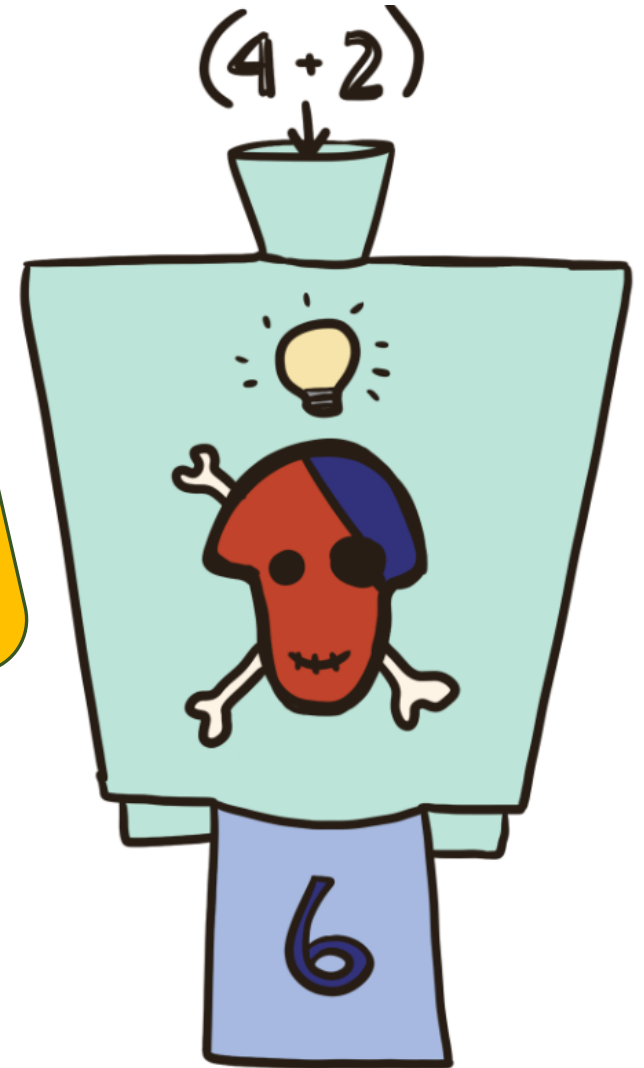
What is the operator / dividing?

Shouldn't / expect two numbers?

Even the

Consider this thingy on the right to be a "black box"
Have you heard the term before?

types, not just numbers!





More On Evaluations

- An expression of the form `<name> = <expression>` tells Pyret to associate the value of `<expression>` with `<name>`. Every time you type `<name>`, Pyret will substitute the value for you:
`x = 5`
`x + 4`
will evaluate to 9.

Creating a definition...



Note there's no output from entering a definition.

It only has a side effect of telling Pyret to associate the name with the value.

```
>>> star(40, "solid", "gold")
```



```
>>> my-star = star(40, "solid", "gold")
```

```
>>> my-star
```





Naming Conventions

- Every programming language has its own conventions for names.
- In Pyret, names are lowercase with words joined by hyphens, e.g.,
`this-is-a-good-name`
`this_makes_bonny_cry`
`thisIsACrimeAgainstPyret`



Naming Conventions (2)

Names must be given a value before being used.

In Pyret, names are *immutable*, which means they can only be defined once.

```
>>> x
The identifier x is unbound:
interactions://1:0:0-0:1
1 | x
It is used but not previously defined.

>>> x = 3
>>> x
3
>>> x = 4
The declaration of x shadows a previous declaration of x

>>> x
3
>>> |
```

Let's try drawing ~~something~~ an eyeball



The screenshot shows a Jupyter Notebook interface. At the top, there is a toolbar with a skull and crossbones icon, a 'View' dropdown, a 'Connect to Google Drive' button, a blue 'Run' button, and a grey 'Stop' button. The main area is split into two panes. The left pane contains a code editor with the following Python code:

```
1 use context essentials2021
2
3 a = ellipse(65, 115, "solid", "black")
4 b = ellipse(50, 100, "solid", "white")
5 eyeball = overlay(b, a)
6
7 pupil = ellipse(15, 25, "solid", "black")
8 overlay(pupil, eyeball)
```

The right pane shows the output of the code, which is a simple drawing of an eyeball. It consists of a large black outline for the eye, a smaller white oval inside for the iris, and a small black dot for the pupil. Below the drawing, there are three right-pointing chevrons (>>>>) indicating the end of the execution.

Programming as a guest.

Let's try drawing ~~something an eyeball~~ 2 eyeballs!



View

Connect to Google Drive

Run

Stop

```
1 use context essentials2021
2 a = ellipse(65, 115, "solid", "black")
3 b = ellipse(50, 100, "solid", "white")
4 eyeball = overlay(b, a)
5
6 pupil = ellipse(15, 25, "solid", "black")
7 #overlay(pupil, eyeball)
8
9 overlay-xy(pupil, eyeball)
10
```

This application expression errored:

[definitions://:8:0-8:26](#)

```
9 overlay-xy(pupil, eyeball)
```

2 arguments were passed to the **operator**.

The **operator** evaluated to a function accepting 4 parameters.


An application expression expects the number of parameters and **arguments** to be the same.

[\(Show program evaluation trace...\)](#)

>>>


Whoops! Whoopsie! Don't forget documentation!



▼  View Connect to Google Drive

Run Stop

```
1 use context essentials2021
2 a = ellipse(65, 115, "solid", "black")
3 b = ellipse(50, 100, "solid", "white")
4 eyeball = overlay(b, a)
5
6 pupil = ellipse(15, 25, "solid", "black")
7 #overlay(pupil, eyeball)
8
9 #overlay-xy(pupil,-35, -60, eyeball)
10 left-eyeball = overlay-xy(pupil,-35, -60,
11 eyeball)
12 right-eyeball = flip-horizontal(left-eyeball)
13 beside(left-eyeball, right-eyeball)
```



>>>

Final Thoughts on the eyeballs



- As you build up more complex images from simpler ones, you're following a core idea called:

COMPOSITION.

- Programs are always built of smaller programs that do parts of the larger task you want to perform.
- We'll use composition throughout this course



Next: What does this code do?

- # Create the head: a yellow circle with black border
- base = circle(50, "solid", "yellow")
- base-border = circle(53, "solid", "black")
- head = overlay(base, base-border)
- # Create pair of eyes, using a square as a spacer
- eye = circle(9, "solid", "blue")
- eye-spacer = square(12, "solid", "yellow")
- one-eye-with-space = beside(eye, eye-spacer)
- eyes = beside(one-eye-with-space, eye)
- # Add a mouth to the eyes to make a face
- mouth = ellipse(30, 15, "solid", "red")
- mouth-spacer = rectangle(30, 15, "solid", "yellow")
- eyes-with-mouth-space = above(eyes, mouth-spacer)
- face = above(eyes-with-mouth-space, mouth)
- # Put the face on the head
- emoji = overlay-align("center", "center", face, head)
- emoji

Too slow: This code makes a smiley emoji




▼ View Connect to Google Drive Run Stop

```
1 use context essentials2021
2
3 # Create the head: a yellow circle with black
  border
4 base = circle(50, "solid", "yellow")
5 base-border = circle(53, "solid", "black")
6 head = overlay(base, base-border)
7 # Create pair of eyes, using a square as a
  spacer
8 eye = circle(9, "solid", "blue")
9 eye-spacer = square(12, "solid", "yellow")
10 one-eye-with-space = beside(eye, eye-spacer)
11 eyes = beside(one-eye-with-space, eye)
12 # Add a mouth to the eyes to make a face
13 mouth = ellipse(30, 15, "solid", "red")
14 mouth-spacer = rectangle(30, 15, "solid",
  "yellow")
15 eyes-with-mouth-space = above(eyes, mouth-
  spacer)
16 face = above(eyes-with-mouth-space, mouth)
17 # Put the face on the head
18 emoji = overlay-align("center", "center",
  face, head)
19 emoji
```


This also makes a smiley emoji



▼  View Connect to Google Drive

Run Stop

```
1 use context essentials2021
2
3 # Create the head: a yellow circle with black border
4 base = circle(50, "solid", "yellow")
5 head = overlay(base, circle(53, "solid", "black"))
6 # Create a pair of eyes, using a square as a spacer
7 eye = circle(9, "solid", "blue")
8 eyes =
9 beside(
10 eye,
11 beside(
12 square(12, "solid", "yellow"), # eye spacer
13 eye))
14 # Add a mouth to the eyes to make a face
15 mouth = ellipse(30, 15, "solid", "red")
16 face =
17 above(
18 eyes,
19 above(
20 rectangle(30, 15, "solid", "yellow"), # mouth spacer
21 mouth))
22 # Put the face on the head
23 emoji = overlay-align("center", "center", face, head)
24 emoji
25
```



>>>



Which version is “better?”

- The first set of code may seem easier to understand. At first.
- As we get more involved working with structured data, writing code like the second slide will be more useful:
 - The structure of well written program tends to reflect the structure of the data you are working with.

• The structure of well written program tends to reflect the structure of the data you are working with.



Eyeball code: Copy From

- `a = ellipse(65, 115, "solid", "black")`
- `b = ellipse(50, 100, "solid", "white")`
- `eyeball = overlay(b, a)`

- `pupil = ellipse(15, 25, "solid", "black")`
- `#overlay(pupil, eyeball)`

- `#overlay-xy(pupil,-35, -60, eyeball)`
- `left-eyeball = overlay-xy(pupil,-35, -60, eyeball)`
- `right-eyeball = flip-horizontal(left-eyeball)`
- `beside(left-eyeball, right-eyeball)`

2nd set of emoji code: Copy From



```
# Create the head: a yellow circle with black border
base = circle(50, "solid", "yellow")
head = overlay(base, circle(53, "solid", "black"))
# Create a pair of eyes, using a square as a spacer
eye = circle(9, "solid", "blue")
eyes =
beside(
eye,
beside(
square(12, "solid", "yellow"), # eye spacer
eye))
# Add a mouth to the eyes to make a face
mouth = ellipse(30, 15, "solid", "red")
face =
above(
eyes,
above(
rectangle(30, 15, "solid", "yellow"), # mouth spacer
mouth))
# Put the face on the head
emoji = overlay-align("center", "center", face, head)
emoji
```