Expressions, Values, and Names

23 January 2024
Where are we?
A *program* instructs a computer to do something.

For the computer to carry out these instructions, they need to be very specific.

But programs also need to be understood by people, so they need to be readable!
We write a program in a *programming language* and we run it in a *programming environment*. 
use context essentials2021
This tells Pyret to include useful functions, like the ones we used for images.
Which pane would I use if…
Which pane would I use if…

I want to see if I can make a blue circle?
Which pane would I use if...

I want to see if I can make a blue circle?

I want to define my-shape as a blue circle and use it later in my code?
Which pane would I use if…

I want to see if I can make a blue circle?
I want to define `my-shape` as a blue circle and use it later in my code?
I want to see if Pyret will accept this: `print "5"`?
Which pane would I use if…

I want to see if I can make a blue circle?

I want to define my-shape as a blue circle and use it later in my code?

I want to see if Pyret will accept this: print "5"?

I want to start my assignment now and finish it later?
Starting to program
We're trying to make sense of the problem.

We start with the *data* before we dive in to try to *do* it.
We might want to compute the heights of the stripes from the overall flag dimensions, which means we need to write programs over *numbers*.

We need a way to describe *colors* to our program.

We need a way to create images based on simple *shapes* of different colors.
We might want to compute the heights of the stripes from the overall flag dimensions, which means we need to write programs over *numbers*.

We need a way to describe *colors* to our program.

We need a way to create images based on simple *shapes* of different colors.
An individual number like 5 is a *value* – it can’t be computed any further.
\((3 + 4) \times (5 + 1)\) is an \textit{expression} – a computation that produces an answer.

A program just consists of one or more computations you want to run.
Reading this expression errored:

1 3 + 4 * 5 + 1

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 + 1)

42
>>> num-min(5, 9)
5
We might want to compute the heights of the stripes from the overall flag dimensions, which means we need to write programs over *numbers*.

We need a way to describe *colors* to our program.

We need a way to create images based on simple *shapes* of different colors.
Names can be given as *text strings*, e.g., "blue".
We might want to compute the heights of the stripes from the overall flag dimensions, which means we need to write programs over *numbers*.

We need a way to describe *colors* to our program.

We need a way to create images based on simple *shapes* of different colors.
include image
You only need to type this if you haven’t pressed “Run” to get the image functions from the context line in the definitions pane.
You only need to type this if you haven’t pressed “Run” to get the image functions from the context line in the definitions pane.

```python
>>> include image

>>> circle(50, "solid", "red")
```
You only need to type this if you haven’t pressed “Run” to get the image functions from the context line in the definitions pane.

```
>>> include image

>>> circle(50, "solid", "red")
```
We can manipulate images much like we can manipulate numbers.

- Numbers can be added, subtracted, etc.
- Images can be overlaid, rotated, flipped, etc.
Evaluation
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!
Operations may only work on certain types of data!
When we write complex expressions, Pyret evaluates them from the inside out:

\[ 7 + \left( 6 \div (1 + 1) \right) \]
When we write complex expressions, Pyret evaluates them from the inside out:

\[ 7 + \frac{6}{1+1} \]
\[ \rightarrow 7 + \frac{6}{2} \]
When we write complex expressions, Pyret evaluates them from the inside out:

\[
7 + (6 / (1 + 1)) \\
\rightarrow 7 + (6 / 2) \\
\rightarrow 7 + 3
\]
When we write complex expressions, Pyret evaluates them from the inside out:

\[
7 + \frac{6}{1 + 1}
\rightarrow 7 + \frac{6}{2}
\rightarrow 7 + 3
\rightarrow 10
\]
When we write complex expressions, Pyret evaluates them from the inside out:

```
beside(
    circle(10, "solid", "red"),
    circle(10, "solid", "blue"))
```

→ \( \text{beside}(\text{circle}(10, "solid", "red"), \text{circle}(10, "solid", "blue")) \)

→ \( \text{beside}(\text{circle}(10, "solid", "red"), \text{circle}(10, "solid", "blue")) \)

→ \( \text{beside}(\text{circle}(10, "solid", "red"), \text{circle}(10, "solid", "blue")) \)

→ \( \text{circle}(10, "solid", "red") \text{ beside } \text{circle}(10, "solid", "blue") \)
What’s in a name?
>>> $x = 5$
There's no output from entering a definition.
There's no output from entering a definition.

It only has the side effect of telling Pyret to associate the name with the value in the program directory.
\[ x = 5 \]

\[ x \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>5</td>
</tr>
</tbody>
</table>
>>> x = 5
>>> x
5

When you use the name later, Pyret looks it up in the directory and substitutes the value it finds.
```python
>>> fname = "Grace"
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>fname</td>
<td>&quot;Grace&quot;</td>
</tr>
</tbody>
</table>
```
>>> fname = "Grace"
>>> lname = "Hopper"
```

```
Directory

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>fname</td>
<td>&quot;Grace&quot;</td>
</tr>
<tr>
<td>lname</td>
<td>&quot;Hopper&quot;</td>
</tr>
</tbody>
</table>
```
```python
>>> fname = "Grace"
>>> lname = "Hopper"
>>> fname + " " + lname
```

```
Directory

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>fname</td>
<td>&quot;Grace&quot;</td>
</tr>
<tr>
<td>lname</td>
<td>&quot;Hopper&quot;</td>
</tr>
</tbody>
</table>
```
```python
>>> fname = "Grace"
>>> lname = "Hopper"
>>> fname + " " + lname
'Grace' + " " + lname
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>fname</td>
<td>&quot;Grace&quot;</td>
</tr>
<tr>
<td>lname</td>
<td>&quot;Hopper&quot;</td>
</tr>
</tbody>
</table>

Directory
```python
>>> fname = "Grace"
>>> lname = "Hopper"
>>> fname + " " + lname
→ "Grace" + " " + lname
→ "Grace " + lname
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>fname</td>
<td>&quot;Grace&quot;</td>
</tr>
<tr>
<td>lname</td>
<td>&quot;Hopper&quot;</td>
</tr>
</tbody>
</table>
```python
>>> fname = "Grace"

>>> lname = "Hopper"

>>> fname + " " + lname

"Grace " + lname

"Grace " + "Hopper"
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>fname</td>
<td>&quot;Grace&quot;</td>
</tr>
<tr>
<td>lname</td>
<td>&quot;Hopper&quot;</td>
</tr>
</tbody>
</table>
```python
>>> fname = "Grace"
>>> lname = "Hopper"
>>> fname + " " + lname
  "Grace" + " " + lname
  "Grace " + lname
  "Grace " + "Hopper"
  "Grace Hopper"
```

### Directory

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>fname</td>
<td>&quot;Grace&quot;</td>
</tr>
<tr>
<td>lname</td>
<td>&quot;Hopper&quot;</td>
</tr>
</tbody>
</table>
Names must be given a value before being used.

In Pyret, names are *immutable*, which means they can only be defined once.
Names must be given a value before being used.

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

The following is silly, but legal:

```python
>>> five = 6
>>> five
6
>>> six = 5
>>> six
5
```
Several constants may have the same value:

```plaintext
  seven = 7
  seven
  7
  septem = 7
  septem
  7
```
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
```
Concept check
We can define the names

\[
\begin{align*}
width &= 400 \\
height &= 600
\end{align*}
\]

Now if we write

\[
width \times height
\]

it gets evaluated:

\[
\begin{align*}
\rightarrow 400 &\times height \\
\rightarrow 400 &\times 600 \\
\rightarrow 240000
\end{align*}
\]
What if we use another name?

```
width  = 400
height = 600
area  = width * height
```

Does Pyret associate the name `area` with the expression `width * height` or with the number `240000`?
Exercise
xeyes
xeyes
```python
use context essentials2021

```
```python
use context essentials2021
b = ellipse(65, 115, "solid", "black")
w = ellipse(50, 100, "solid", "white")
eyeball = overlay(w, b)
```
use context essentials2021

b = ellipse(65, 115, "solid", "black")
w = ellipse(50, 100, "solid", "white")

eyeball = overlay(w, b)

>>> eyeball

>>> ellipse(20, 40, "solid", "black")

>>> ellipse(15, 25, "solid", "black")
use context essentials2021
b = ellipse(65, 115, "solid", "black")
w = ellipse(50, 100, "solid", "white")
eyeball = overlay(w, b)
pupil = ellipse(15, 25, "solid", "black")
overlay(pupil, eyeball)
ais2021,
    "solid", "black")
    , (25, "solid", "black")
    , (b)
6 Glossary

overlay

overlay (from image)
overlay-align (from image)
overlay-onto-offset (from image)
overlay-xy (from image)
Overlaying Images
overlay :: {
    img1 :: Image,
    img2 :: Image
} -> Image

Constructs a new image where img1 overlays img2. The two images are aligned at their pinholes, so overlay(img1, img2) behaves like overlay-align("pinhole", "pinhole", img1, img2).

Examples:

```plaintext
>>> overlay(rectangle(30, 60, "solid", "orange"), ellipse(60, 30, "solid", "purple"))
```

overlay-align :: {
    place-x :: XPlace,
    place-y :: YPlace,
    img1 :: Image,
    img2 :: Image
} -> Image

Overlays img1 on img2 like overlay, but uses place-x and place-y to determine the alignment point in each image. A call to overlay-align(place-x, place-y, img1, img2) behaves the same as overlay-onto-offset(img1, place-x, place-y, 0, 0, img2, place-x, place-y)

Examples:

```plaintext
>>> overlay-align("left", "bottom",
    square(30, "solid", "bisque"), square(50, "solid", "dark-green"))
```
use context essentials2021

b = ellipse(65, 115, "solid", "black")
w = ellipse(50, 100, "solid", "white")
eyeball = overlay(w, b)
pupil = ellipse(15, 25, "solid", "black")

overlay-align("right", "bottom", pupil, eyeball)
use context essentials2021

b = ellipse(65, 115, "solid", "black")
w = ellipse(50, 100, "solid", "white")

eyeball = overlay(w, b)
pupil = ellipse(15, 25, "solid", "black")

overlay-xy(pupil, -35, -60, eyeball)
use context essentials2021

b = ellipse(65, 115, "solid", "black")
w = ellipse(50, 100, "solid", "white")
eyeball = overlay(w, b)
pupil = ellipse(15, 25, "solid", "black")
left-eye = overlay-xy(pupil, -35, -60, eyeball)
right-eye = flip-horizontal(left-eye)
beside(left-eye, right-eye)
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.
Organizing a program with names
Let’s consider three programs that all draw this (beautiful, nuanced) emoji:
# 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
# Create the head: a yellow circle with black border

```python
base = circle(50, "solid", "yellow")
head = overlay(base, circle(53, "solid", "black"))
```

# Create a pair of eyes, using a square as a spacer

```python
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

```python
mouth = ellipse(30, 15, "solid", "red")
face =
    above(
        eyes,
        above(
            rectangle(30, 15, "solid", "yellow"),  # mouth spacer
            mouth))
```

# Put the face on the head

```python
emoji = overlay-align("center", "center", face, head)
emoji
```
overlay-align("center", "center",
above(
    beside(
        circle(9, "solid", "blue"), # eye
        beside(
            square(12, "solid", "yellow"), # eye spacer
            circle(9, "solid", "blue"))), # eye
    above(
        rectangle(30, 15, "solid", "yellow"), # mouth spacer
        ellipse(30, 15, "solid", "red"))), # mouth
    overlay(circle(50, "solid", "yellow"), # base
        circle(53, "solid", "black"))) # head border
All three programs generate the same image.

Which one seems easiest to read and understand?
# 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
overlay-align("center", "center",
    above(
        beside(
            circle(9, "solid", "blue"), # eye
        beside(
            square(12, "solid", "yellow"), # eye spacer
            circle(9, "solid", "blue"))), # eye
    above(
        rectangle(30, 15, "solid", "yellow"), # mouth spacer
        ellipse(30, 15, "solid", "red"))), # mouth
    overlay(circle(50, "solid", "yellow"), # base
        circle(53, "solid", "black"))) # head border
overlay-align("center", "center",
    above(
        beside(
            circle(9, "solid", "blue"),
            beside(
                square(12, "solid", "yellow"),
                circle(9, "solid", "blue"))),
        above(
            rectangle(30, 15, "solid", "yellow"),
            ellipse(30, 15, "solid", "red")),
        overlay(circle(50, "solid", "yellow"),
            circle(53, "solid", "black"))))
# 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
Beginning programmers tend to write code more like the first or third examples, naming everything or nothing.

As we get more into working with structured data, writing code like the second example will be useful, as the structure of a well-written program tends to reflect the structure of the data you are working with.
“Programs must be written for people to read, and only incidentally for machines to execute.”

Hal Abelson & Gerald Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, 1979
Acknowledgments

This class incorporates material from:

Kathi Fisler, Brown University
Jason Waterman, Vassar College