Introduction to Programming

The Editor is a software program we use to write Code. Our Editor allows us to experiment with Code on the right-hand side, in the Interactions Area. For Code that we want to keep, we can put it on the left-hand side in the Definitions Area. Clicking the "Run" button causes the computer to re-read everything in the Definitions Area and erase anything that was typed into the Interactions Area.

Data Types

Programming languages involve different data types, such as Numbers, Strings, Booleans, and even Images.

- Numbers are values like 1, 0.4, 1/3, and -8261.003.
  - Numbers are usually used for quantitative data and other values are usually used as categorical data.
  - In Pyret, any decimal must start with a 0. For example, 0.22 is valid, but .22 is not.
- Strings are values like "Emma", "Rosanna", "Jen and Ed", or even "08/28/1980".
  - All strings must be surrounded in quotation marks.
- Booleans are either true or false.

All values evaluate to themselves. The program 42 will evaluate to 42, the String "Hello" will evaluate to "Hello", and the Boolean false will evaluate to false.

Operators

Operators (like +, -, *, <, etc.) work the same way in Pyret that they do in math.

- Operators are written between values, for example: 4 + 2.
- In Pyret, operators must always have a space around them. 4 + 2 is valid, but 4+2 is not.
- If an expression has different operators, parentheses must be used to show order of operations. 4 + 2 + 6 and 4 + (2 * 6) are valid, but 4 + 2 * 6 is not.

Applying Functions

Applying functions works much the way it does in math. Every function has a name, takes some inputs, and produces some output. The function name is written first, followed by a list of arguments in parentheses.

- In math this could look like $f(5)$ or $g(10, 4)$.
- In Pyret, these examples would be written as $f(5)$ and $g(10, 4)$.
- Applying a function to make images would look like $\text{star}(50, "solid", "red")$.
- There are many other functions, for example num-sqr, num-sqrt, triangle, square, string-repeat, etc.

Functions have contracts, which help explain how a function should be used. Every contract has three parts:

- The Name of the function - literally, what it’s called.
- The Domain of the function - what types of values the function consumes, and in what order.
- The Range of the function - what type of value the function produces.
Numbers and Strings

Make sure you’ve loaded the code.pyret.org, (CPO) editor, clicked “Run”, and are working in the Interactions Area.

Numbers

1) Try typing 42 into the Interactions Area and hitting “Enter”. What is the largest number the editor can handle?

2) Try typing 8.5. Then try typing .5. Then try clicking on the answer. Experiment with other decimals. Explain what you understand about how decimals work in this programming language.

3) What happens if you try a fraction like 1/3?

4) Try writing negative integers, fractions and decimals.

Strings

String values are always in quotes.

5) Is 42 the same as “42”? Why or why not? Write your answer below:

6) Try typing your name (in quotes!).

7) Try typing a sentence like “I’m excited to learn to code!” (in quotes!).

8) Try typing your name with the opening quote, but without the closing quote. Read the error message!

9) Now try typing your name without any quotes. Read the error message!

10) Explain what you understand about how strings work in this programming language.

Operators

11) Just like math, Pyret has operators like +, -, *, and / . Try typing in 4 + 2, and then 4 + 2 (without the spaces). What can you conclude from this?

12) Type in the following expressions, one at a time: 4 + 2 + 6, 4 + 2 * 6, 4 + (2 * 6). What do you notice?

13) Try typing in 4 + "cat", and then "dog" + "cat". What can you conclude from this?
Booleans

Boolean-producing expressions are yes-or-no questions and will always evaluate to either \textbf{true} ("yes") or \textbf{false} ("no"). What will each of the expressions below evaluate to? Write down your prediction in the blanks provided and then type the code into the interactions area to see what it returns.

<table>
<thead>
<tr>
<th>Prediction:</th>
<th>Computer Returns:</th>
<th>Prediction:</th>
<th>Computer Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) (3 \leq 4)</td>
<td></td>
<td>2) (\text{&quot;a&quot; &gt; &quot;b&quot;})</td>
<td></td>
</tr>
<tr>
<td>3) (3 \geq 2)</td>
<td></td>
<td>4) (\text{&quot;a&quot; &lt; &quot;b&quot;})</td>
<td></td>
</tr>
<tr>
<td>5) (2 &lt; 4)</td>
<td></td>
<td>6) (\text{&quot;a&quot; == &quot;b&quot;})</td>
<td></td>
</tr>
<tr>
<td>7) (5 &gt;= 5)</td>
<td></td>
<td>8) (\text{&quot;a&quot; &lt;&gt; &quot;a&quot;})</td>
<td></td>
</tr>
<tr>
<td>9) (4 &gt; 6)</td>
<td></td>
<td>10) (\text{&quot;a&quot; &gt;= &quot;a&quot;})</td>
<td></td>
</tr>
<tr>
<td>11) (\text{&quot;a&quot; &lt;&gt; &quot;b&quot;})</td>
<td></td>
<td>12) (\text{&quot;a&quot; &lt;= &quot;b&quot;})</td>
<td></td>
</tr>
</tbody>
</table>

13) In your own words, describe what \(<\) does.

14) In your own words, describe what \(\geq\) does.

15) In your own words, describe what \(<>\) does.

<table>
<thead>
<tr>
<th>Prediction:</th>
<th>Computer Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>16) string-contains(&quot;catnap&quot;, &quot;cat&quot;)</td>
<td></td>
</tr>
<tr>
<td>17) string-contains(&quot;cat&quot;, &quot;catnap&quot;)</td>
<td></td>
</tr>
</tbody>
</table>

18) How many \textbf{Numbers} are there in the entire universe? 
19) How many \textbf{Strings} are there in the entire universe? 
20) How many \textbf{Images} are there in the entire universe? 
21) How many \textbf{Booleans} are there in the entire universe?
Applying Functions

Type this line of code into the interactions area and hit "Enter":

```
triangle(50, "solid", "red")
```

1. What is the name of this function?
2. What did the expression evaluate to?
3. How many arguments does \texttt{triangle} expect?
4. What data type does the \texttt{triangle} function produce?
   (Numbers? Strings? Booleans?)

Catching Bugs

The following lines of code are all BUGGY! Read the code and the error messages to identify the mistake.

5) \texttt{triangle(20, "solid" "red")}

Pyret didn't understand your program around

```
triangle(20, "solid" "red")
```

Can you spot the mistake?

6) \texttt{triangle(20, "solid")}

This \texttt{application expression} errored:

```
triangle(20, "solid")
```

2 \texttt{arguments} were passed to the \texttt{operator}. The \texttt{operator} evaluated to a function accepting 3 parameters. An \texttt{application expression} expects the number of parameters and \texttt{arguments} to be the same.

Can you spot the mistake?

7) \texttt{triangle(20, 10, "solid", "red")}

This \texttt{application expression} errored:

```
triangle(20, 10, "solid", "red")
```

4 \texttt{arguments} were passed to the \texttt{operator}. The \texttt{operator} evaluated to a function accepting 3 parameters. An \texttt{application expression} expects the number of parameters and \texttt{arguments} to be the same.

Can you spot the mistake?

8) \texttt{triangle (20, "solid", "red")}

Pyret thinks this code is probably a function call:

```
triangle (20, "solid", "red")
```

Function calls must not have space between the \texttt{function expression} and the \texttt{arguments}.

Can you spot the mistake?
Domain and Range

My Definition

Examples

Domain

Facts & Characteristics

Non-examples

Range

My Definition

Examples

Facts & Characteristics

Non-examples
Consider the following contract:

```
is-beach-weather :: Number, String -> Boolean
```

1) What is the **Name** of this function?  
2) How many arguments are in this function's **Domain**?  
3) What is the **type** of this function's **first argument**?  
4) What is the **type** of this function's **second argument**?  
5) What is the **Range** of this function?  

6) Circle the expression below that shows the correct application of this function, based on its contract.

   A. `is-beach-weather(70, 90)`  
   B. `is-beach-weather(80, 100, "cloudy")`  
   C. `is-beach-weather("sunny", 90)`  
   D. `is-beach-weather(90, "stormy weather")`  

Consider the following contract:

```
cylinder :: Number, Number, String -> Image
```

7) What is the **Name** of this function?  
8) How many arguments are in this function's **Domain**?  
9) What is the **type** of this function's **first argument**?  
10) What is the **type** of this function's **second argument**?  
11) What is the **type** of this function's **third argument**?  
12) What is the **Range** of this function?  

13) Circle the expression below that shows the correct application of this function, based on its contract.

   A. `cylinder("red", 10, 60)`  
   B. `cylinder(30, "green")`  
   C. `cylinder(10, 25, "blue")`  
   D. `cylinder(14, "orange", 25)`
Matching Expressions and Contracts

Match the contract (left) with the expression described by the function being used (right).

<table>
<thead>
<tr>
<th>Contract</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td># make-id :: String, Number -&gt; Image</td>
<td>A make-id(&quot;Savannah&quot;, &quot;Lopez&quot;, 32)</td>
</tr>
<tr>
<td># make-id :: String, Number, String -&gt; Image</td>
<td>B make-id(&quot;Pilar&quot;, 17)</td>
</tr>
<tr>
<td># make-id :: String -&gt; Image</td>
<td>C make-id(&quot;Akemi&quot;, 39, &quot;red&quot;)</td>
</tr>
<tr>
<td># make-id :: String, String -&gt; Image</td>
<td>D make-id(&quot;Raissa&quot;, &quot;McCracken&quot;)</td>
</tr>
<tr>
<td># make-id :: String, String, Number -&gt; Image</td>
<td>E make-id(&quot;von Einsiedel&quot;)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Contract</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td># is-capital :: String, String -&gt; Boolean</td>
<td>A show-pop(&quot;Juneau&quot;, &quot;AK&quot;, 31848)</td>
</tr>
<tr>
<td># is-capital :: String, String, String -&gt; Boolean</td>
<td>B show-pop(&quot;San Juan&quot;, 395426)</td>
</tr>
<tr>
<td># show-pop :: String, Number -&gt; Image</td>
<td>C is-capital(&quot;Accra&quot;, &quot;Ghana&quot;)</td>
</tr>
<tr>
<td># show-pop :: String, String, Number -&gt; Image</td>
<td>D show-pop(3751351, &quot;Oklahoma&quot;)</td>
</tr>
<tr>
<td># show-pop :: Number, String -&gt; Number</td>
<td>E is-capital(&quot;Albany&quot;, &quot;NY&quot;, &quot;USA&quot;)</td>
</tr>
</tbody>
</table>
### Using Contracts

Use the contracts to write expressions to generate images similar to those pictured.

**ellipse**: Number, Number, String, String -> Image

<table>
<thead>
<tr>
<th>Image</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="ellipse.png" alt="Ellipse" /></td>
<td>What changes with the first number?</td>
</tr>
<tr>
<td><img src="ellipse.png" alt="Ellipse" /></td>
<td>What about the shape changes with the second Number?</td>
</tr>
<tr>
<td><img src="ellipse.png" alt="Ellipse" /></td>
<td>Write an expression using <strong>ellipse</strong> to produce a circle.</td>
</tr>
</tbody>
</table>

**regular-polygon**: Number, Number, String, String -> Image

<table>
<thead>
<tr>
<th>Image</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="pentagon.png" alt="Regular Pentagon" /></td>
<td>What changes with the first Number?</td>
</tr>
<tr>
<td><img src="pentagon.png" alt="Regular Pentagon" /></td>
<td>What about the shape changes with the second Number?</td>
</tr>
<tr>
<td><img src="pentagon.png" alt="Regular Pentagon" /></td>
<td>Use <strong>regular-polygon</strong> to write an expression for a square!</td>
</tr>
<tr>
<td><img src="pentagon.png" alt="Regular Pentagon" /></td>
<td>How would you describe a <strong>regular polygon</strong> to a friend?</td>
</tr>
</tbody>
</table>
1) What kind of triangle does the `triangle` function produce? 

There are lots of other kinds of triangles! And Pyret has lots of other functions that make triangles!

```
triangle :: (size::Number, style::String, color::String) -> Image
right-triangle :: (base::Number, height::Number, style::String, color::String) -> Image
isosceles-triangle :: (leg::Number, angle::Number, style::String, color::String) -> Image
```

2) Why do you think `triangle` only needs one number, while `right-triangle` and `isosceles-triangle` need two numbers and `triangle-sas` needs three?

3) Write `right-triangle` expressions for the images below. One argument for each should be `100`.

```
\[ \text{Image} \]
```

```
\[ \text{Image} \]
```

4) What do you think the numbers in `right-triangle` represent?

5) Write `isosceles-triangle` expressions for the images below. One argument for each should be `100`.

```
\[ \text{Image} \]
```

```
\[ \text{Image} \]
```

6) What do you think the numbers in `isosceles-triangle` represent?

7) Write 2 expressions that would build right-isosceles triangles. Use `right-triangle` for one expression and `isosceles-triangle` for the other expression.

```
\[ \text{Image} \]
```

```
\[ \text{Image} \]
```
Using the detailed contract above, match each image to the expression that describes it.

<table>
<thead>
<tr>
<th>Image</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>radial-star(5, 50, 200, &quot;solid&quot;, &quot;black&quot;)</td>
</tr>
<tr>
<td>2</td>
<td>radial-star(7, 100, 200, &quot;solid&quot;, &quot;black&quot;)</td>
</tr>
<tr>
<td>3</td>
<td>radial-star(7, 100, 200, &quot;outline&quot;, &quot;black&quot;)</td>
</tr>
<tr>
<td>4</td>
<td>radial-star(10, 150, 200, &quot;solid&quot;, &quot;black&quot;)</td>
</tr>
<tr>
<td>5</td>
<td>radial-star(10, 20, 200, &quot;solid&quot;, &quot;black&quot;)</td>
</tr>
<tr>
<td>6</td>
<td>radial-star(100, 20, 200, &quot;solid&quot;, &quot;black&quot;)</td>
</tr>
<tr>
<td>7</td>
<td>radial-star(100, 100, 200, &quot;outline&quot;, &quot;black&quot;)</td>
</tr>
</tbody>
</table>
For each function composition diagrammed below, translate it into the equivalent Circle of Evaluation for Order of Operations. Then write expressions for both versions of the Circles of Evaluation, and evaluate them for \(x = 4\). The first one has been completed for you.

<table>
<thead>
<tr>
<th>Function Composition</th>
<th>Order of Operations</th>
<th>Translate &amp; Evaluate</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f(x) = 3x)</td>
<td>(g(x) = x + 6)</td>
<td>(h(x) = x - 1)</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{Composition:} & \quad h(g(f(x))) \\
\text{Operations:} & \quad ((3 \cdot x) + 6) - 1 \\
\text{Evaluate for } x = 4 & \quad h(g(f(4))) = 17
\end{align*}
\]

\[
\begin{align*}
\text{Composition:} & \\
\text{Operations:} & \\
\text{Evaluate for } x = 4 & \\
\end{align*}
\]

\[
\begin{align*}
\text{Composition:} & \\
\text{Operations:} & \\
\text{Evaluate for } x = 4 & \\
\end{align*}
\]

\[
\begin{align*}
\text{Composition:} & \\
\text{Operations:} & \\
\text{Evaluate for } x = 4 & \\
\end{align*}
\]

\[
\begin{align*}
\text{Composition:} & \\
\text{Operations:} & \\
\text{Evaluate for } x = 4 & \\
\end{align*}
\]
1) Draw a Circle of Evaluation and write the Code for a solid, green star, size 50.

Circle of Evaluation:

<table>
<thead>
<tr>
<th>Code:</th>
</tr>
</thead>
</table>

Using the star described above as the original, draw the Circles of Evaluation and write the Code for each exercise below.

<table>
<thead>
<tr>
<th>2) A solid, green star, that is triple the size of the original (using scale)</th>
<th>3) A solid, green star, that is half the size of the original (using scale)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle of Evaluation:</td>
<td>Circle of Evaluation:</td>
</tr>
<tr>
<td>Code:</td>
<td>Code:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4) A solid, green star of size 50 that has been rotated 45 degrees counter-clockwise</th>
<th>5) A solid, green star that is 3 times the size of the original and has been rotated 45 degrees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle of Evaluation:</td>
<td>Circle of Evaluation:</td>
</tr>
<tr>
<td>Code:</td>
<td>Code:</td>
</tr>
</tbody>
</table>
You’ll be investigating these functions with your partner:

- `text :: String, Number, String -> Image`
- `flip-horizontal :: Image -> Image`
- `flip-vertical :: Image -> Image`
- `frame :: Image -> Image`
- `above :: Image, Image -> Image`
- `beside :: Image, Image -> Image`

1) In the editor, write the code to make an image of your name in big letters in a color of your choosing using `text`. Then draw the Circle of Evaluation and write the Code that will create the image.

**Circle of Evaluation:**

**Code:**

Using the “image of your name” described above as the original, draw the Circles of Evaluation and write the Code for each exercise below. Test your ideas in the editor to make sure they work.

2) The framed ”image of your name”.
   **Circle of Evaluation:**

   **Code:** ________________

3) The ”image of your name” flipped vertically.
   **Circle of Evaluation:**

   **Code:** ________________

4) The ”image of your name” above ”the image of your name” flipped vertically.
   **Circle of Evaluation:**

   **Code:** ________________

5) The ”image of your name” flipped horizontally beside ”the image of your name”.
   **Circle of Evaluation:**

   **Code:** ________________
Function Composition — scale-xy

You’ll be investigating these two functions with your partner:

# scale-xy :: Number, Number, Image -> Image
# overlay :: Image, Images -> Image

<table>
<thead>
<tr>
<th>The Image:</th>
<th>Circle of Evaluation:</th>
<th>Code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhombus</td>
<td>40 90 &quot;solid&quot; &quot;purple&quot;</td>
<td>rhombus(40, 90, &quot;solid&quot;, &quot;purple&quot;)</td>
</tr>
</tbody>
</table>

Starting with the image described above, write the Circles of Evaluation and Code for each exercise below. Be sure to test your code in the editor!

1) A purple rhombus that is stretched 4 times as wide.
   Circle of Evaluation:
   Code: ______________________________

2) A purple rhombus that is stretched 4 times as tall
   Circle of Evaluation:
   Code: ______________________________

3) The tall rhombus overlayed on the wide rhombus.
   Circle of Evaluation:
   Code: ______________________________

★: Overlay a red rhombus onto the last image you made.
   Circle of Evaluation:
   Code: ______________________________
More than one way to Compose an Image!

Read through these 4 expressions and try to picture the images they are composing. If you're not sure what they look like, type them into the interactions area of your editor and see if you can figure out how the code connects to the image.

For each image below, identify 2 expressions that could be used to compose it. The bank of expressions at the top of the page includes one possible option for each image.

1. `rotate(90, rectangle(200, 100, "solid", "black"))`
2. `above(rectangle(200, 100, "solid", "black"), rectangle(100, 50, "solid", "black"))`
3. `scale(0.5, rectangle(600, 200, "solid", "black"))`
4. `overlay(rectangle(100, 200, "solid", "black"), rectangle(100, 200, "solid", "black"), rectangle(100, 200, "solid", "black"))`
Defining Values

In math, we use values like $-98.1, \frac{2}{3}$ and 42. In math, we also use expressions like $1 \times 3, \sqrt{16}$, and $5 - 2$. These evaluate to results, and typing any of them in as code produces some answer.

Math also has definitions. These are different from values and expressions, because they do not produce results. Instead, they simply create names for values, so that those names can be re-used to make the Math simpler and more efficient.

Definitions always have both a name and an expression. The name goes on the left and the value-producing expression goes on the right, separated by an equals sign:

\[
\begin{align*}
x &= 4 \\
y &= 9 + x
\end{align*}
\]

The name is defined to be the result of evaluating the expression. Using the above examples, we get “x is defined to be 4, and y is defined to be 13”. Important: there is no "answer" to a definition, and typing in a definition as code will produce no result.

Notice that definitions can refer to previous definitions. In the example above, the definition of y refers to x. But x, on the other hand, cannot refer to y. Once a value has been defined, it can be used in later expressions.

In Pyret, these definitions are written the exact same way:

Try typing these definitions into the Definitions Area on the left, clicking "Run", and then using them in the Interactions Area on the right.

\[
\begin{align*}
x &= 4 \\
y &= 9 + x
\end{align*}
\]

Just like in math, definitions in our programming language can only refer to previously-defined values.

Here are a few more value definitions. Feel free to type them in, and make sure you understand them.

\[
\begin{align*}
x &= 5 + 1 \\
y &= x \times 7 \\
food &= "Pizza!" \\
dot &= \text{circle}(y, "solid", "red")
\end{align*}
\]
Open the Defining Values Starter File and click run.

1) What do you notice?


2) What do you wonder?


Look at the expressions listed below. Think about what you expect each of them to produce. Then, test them out one at a time in the Interactions Area.

- x
- x + 5
- y - 9
- x * y
- z
- t
- gold-star
- my-name
- swamp
- c

3) What have you learned about defining values?


4) Define at least 2 more variables in the definitions area, click run and test them out. Once you know they’re working, record the code you used below.


Defining Values - Chinese Flag

1) What image do you see repeated in the flag?

2) Highlight or circle all instances of the structure that makes the repeated image in the code below.

3) In the code below, highlight or circle all instances of the expression for that image.

```
put-image(
    rotate(40, star(15, "solid", "yellow")),
    120, 175,
    put-image(
        rotate(80, star(15, "solid", "yellow")),
        140, 150,
        put-image(
            rotate(60, star(15, "solid", "yellow")),
            140, 120,
            put-image(
                rotate(40, star(15, "solid", "yellow")),
                120, 90,
                put-image(scale(3, star(15, "solid", "yellow")),
                           60, 140,
                           rectangle(300, 200, "solid", "red")))
        ))
```

4) Write the code to define a value for the repeated expression.

5) Open the Chinese flag starter file (Pyret) and click Run.
Then type `china` into the interactions area and click Enter.

6) Save a copy of the file, and simplify the flag code using the value you defined. Click Run, and confirm that you still get the same image as the original.

7) Now change the color of all of the stars to black, in both files. Then change the size of the stars.

8) Why is it helpful to define values for repeated images?

---

Challenge:
- This file uses a function we haven’t seen before! What is it? ____________________________
- Can you figure out its contract? *Hint: Focus on the last instance of the function.*
### Why Define Values?

1. Complete the table using the first row as an example.
2. Write the code to define the value of `sunny`.
3. Test your code in the editor and make sure it produces what you would expect it to.

<table>
<thead>
<tr>
<th>Original Circle of Evaluation &amp; Code</th>
<th>→</th>
<th>Code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>use the defined value <code>sunny</code> to simplify</td>
<td>---</td>
<td><code>scale (3, 'sunny')</code></td>
</tr>
<tr>
<td>2) Why define the value of <code>sunny</code>?</td>
<td>---</td>
<td><code>overlay (text('sun', 30, 'black'), radial-star(30, 20, 50, 'solid', 'yellow'))</code></td>
</tr>
<tr>
<td>2) Complete the table using the first row as an example</td>
<td>---</td>
<td><code>frame (radial-star(30, 20, 50, 'solid', 'yellow'))</code></td>
</tr>
</tbody>
</table>

---

### Why Define Values?

1. Complete the table using the first row as an example.
2. Write the code to define the value of `sunny`.
3. Test your code in the editor and make sure it produces what you would expect it to.
Which Value(s) Would it Make Sense to Define?

For each of the images below, identify which element(s) you would want to define before writing code to compose the image. Hint: what gets repeated?

<table>
<thead>
<tr>
<th>Philippines</th>
<th>St. Vincent &amp; the Grenadines</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Flag" /></td>
<td><img src="image2.png" alt="Flag" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Liberia</th>
<th>Republic of Georgia</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3.png" alt="Flag" /></td>
<td><img src="image4.png" alt="Flag" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quebec</th>
<th>South Korea</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image5.png" alt="Flag" /> <img src="image6.png" alt="Flag" /></td>
<td><img src="image7.png" alt="Flag" /></td>
</tr>
</tbody>
</table>
Writing Code using Defined Values

1) On the line below, write the Code to define PRIZE-STAR as a pink, outline star of size 65.

Using the PRIZE-STAR definition from above, draw the Circle of Evaluation and write the Code for each of the exercises. One Circle of Evaluation has been done for you.

<table>
<thead>
<tr>
<th>Exercise</th>
<th>Description</th>
<th>Circle of Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2)</td>
<td>The outline of a pink star that is three times the size of the original (using scale)</td>
<td>scale 3 PRIZE-STAR</td>
</tr>
<tr>
<td>Code:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3)</td>
<td>The outline of a pink star that is half the size of the original (using scale)</td>
<td></td>
</tr>
<tr>
<td>Code:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4)</td>
<td>The outline of a pink star that is rotated 45 degrees (It should be the same size as the original.)</td>
<td></td>
</tr>
<tr>
<td>Code:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5)</td>
<td>The outline of a pink star that is three times as big as the original and has been rotated 45 degrees</td>
<td></td>
</tr>
<tr>
<td>Code:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6)</td>
<td>How does defining values help you as a programmer?</td>
<td></td>
</tr>
</tbody>
</table>

---

22
Estimating Coordinates

Think of the background image as a sheet of graph paper with the origin (0,0) in the bottom left corner. The numbers in `put-image` specify a point on that graph paper, where the center of the top image should be placed.

The width of the rectangle is 300 and the height is 200. The definitions for `dot` and `background` are:

- `dot = circle(50, "solid", "red")`
- `background = rectangle(300, 200, "outline", "black")`

**Estimate:** What coordinates for the `dot` would create each of the following images?

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

```
put-image(dot, _____, background)  put-image(dot, _____, background)
```

```
put-image(dot, _____, background)  put-image(dot, _____, background)
```
Decomposing Flags

Each of the flags below is shown with their width and height. Identify the shapes that make up each flag. Use the flag’s dimensions to estimate the dimensions of the different shapes. Then estimate the x and y coordinates for the point at which the center of each shape should be located on the flag. *Hint: The bottom left corner of each flag is at (0,0) and the top right corner is given by the flag’s dimensions.*

<table>
<thead>
<tr>
<th>Flag</th>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cameroon</td>
<td>(450 x 300)</td>
</tr>
<tr>
<td>Chile</td>
<td>(420 x 280)</td>
</tr>
<tr>
<td>Panama</td>
<td>(300 x 200)</td>
</tr>
<tr>
<td>Norway</td>
<td>(330 x 240)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>shape:</th>
<th>color:</th>
<th>width:</th>
<th>height:</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>shape:</th>
<th>color:</th>
<th>width:</th>
<th>height:</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>shape:</th>
<th>color:</th>
<th>width:</th>
<th>height:</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>shape:</th>
<th>color:</th>
<th>width:</th>
<th>height:</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Defining Functions

Functions can be viewed in multiple representations. You already know one of them: Contracts, which specify the Name, Domain, and Range of a function. Contracts are a way of thinking of functions as a mapping between one set of data and another. For example, a mapping from Numbers to Strings:

```plaintext
f :: Number -> String
```

Another way to view functions is with Examples. Examples are essentially input-output tables, showing what the function would do for a specific input:

In our programming language, we focus on the last two columns and write them as code:

```plaintext
examples:
  f(1) ts 1 + 2
  f(2) ts 2 + 2
  f(3) ts 3 + 2
  f(4) ts 4 + 2
end
```

Finally, we write a formal function definition ourselves. The pattern in the Examples becomes abstract (or "general"), replacing the inputs with variables. In the example below, the same definition is written in both math and code:

```plaintext
f(x) = x + 2
fun f(x): x + 2 end
```

Look for connections between these three representations!

- The function name is always the same, whether looking at the Contract, Examples, or Definition.
- The number of inputs in the Examples is always the same as the number of types in the Domain, which is always the same as the number of variables in the Definition.
- The "what the function does" pattern in the Examples is almost the same in the Definition, but with specific inputs replaced by variables.
Look at each set of examples on the left and circle what is changing from one example to the next. Then, match the examples on the left to the definitions on the right.

<table>
<thead>
<tr>
<th>Examples:</th>
<th>Functions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$f(x)$</td>
</tr>
<tr>
<td>1</td>
<td>(2 \times 1)</td>
</tr>
<tr>
<td>2</td>
<td>(2 \times 2)</td>
</tr>
<tr>
<td>3</td>
<td>(2 \times 3)</td>
</tr>
<tr>
<td>$x$</td>
<td>$f(x)$</td>
</tr>
<tr>
<td>15</td>
<td>(15 - 3)</td>
</tr>
<tr>
<td>25</td>
<td>(25 - 3)</td>
</tr>
<tr>
<td>35</td>
<td>(35 - 3)</td>
</tr>
<tr>
<td>$x$</td>
<td>$f(x)$</td>
</tr>
<tr>
<td>10</td>
<td>(10 + 2)</td>
</tr>
<tr>
<td>15</td>
<td>(15 + 2)</td>
</tr>
<tr>
<td>20</td>
<td>(20 + 2)</td>
</tr>
<tr>
<td>$x$</td>
<td>$f(x)$</td>
</tr>
<tr>
<td>0</td>
<td>(3(0) - 2)</td>
</tr>
<tr>
<td>1</td>
<td>(3(1) - 2)</td>
</tr>
<tr>
<td>2</td>
<td>(3(2) - 2)</td>
</tr>
<tr>
<td>$x$</td>
<td>$f(x)$</td>
</tr>
<tr>
<td>10</td>
<td>(2(10) + 1)</td>
</tr>
<tr>
<td>20</td>
<td>(2(20) + 1)</td>
</tr>
<tr>
<td>30</td>
<td>(2(30) + 1)</td>
</tr>
</tbody>
</table>
Matching Examples and Function Definitions

Highlight the variables in `gt` and label them with the word "size".

```plaintext
examples:
  gt(20) is triangle(20, "solid", "green")
  gt(45) is triangle(45, "solid", "green")
end
fun gt(size): triangle(size, "solid", "green") end
```

Highlight and label the variables in the example lists below. Then, using `gt` as a model, match the examples to their corresponding function definitions.

<table>
<thead>
<tr>
<th>Examples</th>
<th>Definition</th>
</tr>
</thead>
</table>
| `f("solid")` is circle(8, "solid", "red") | 1 A `fun f(s): star(s, "outline", "red") end`
| `f("outline")` is circle(8, "outline", "red") | |

<table>
<thead>
<tr>
<th>Examples</th>
<th>Definition</th>
</tr>
</thead>
</table>
| `f(2)` is `2 + 2` | 2 B `fun f(num): num + num end`
| `f(4)` is `4 + 4` | |
| `f(5)` is `5 + 5` | |

<table>
<thead>
<tr>
<th>Examples</th>
<th>Definition</th>
</tr>
</thead>
</table>
| `f("red")` is circle(7, "solid", "red") | 3 C `fun f(c): star(9, "solid", c) end`
| `f("teal")` is circle(7, "solid", "teal") | |

<table>
<thead>
<tr>
<th>Examples</th>
<th>Definition</th>
</tr>
</thead>
</table>
| `f("red")` is star(9, "solid", "red") | 4 D `fun f(s): circle(8, s, "red") end`
| `f("grey")` is star(9, "solid", "grey") | |
| `f("pink")` is star(9, "solid", "pink") | |

<table>
<thead>
<tr>
<th>Examples</th>
<th>Definition</th>
</tr>
</thead>
</table>
| `f(3)` is star(3, "outline", "red") | 5 E `fun f(c): circle(7, "solid", c) end`
<p>| <code>f(8)</code> is star(8, &quot;outline&quot;, &quot;red&quot;) | |</p>
<table>
<thead>
<tr>
<th>Contract</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>A # f :: Number -&gt; Number</td>
<td>f(5) is 5/2</td>
</tr>
<tr>
<td>B # f :: String -&gt; Image</td>
<td>f(&quot;pink&quot;, 5) is star(5, &quot;solid&quot;, &quot;pink&quot;)</td>
</tr>
<tr>
<td>C # f :: Number -&gt; Image</td>
<td>f(&quot;Hi!&quot;) is text(&quot;Hi!&quot;, 50, &quot;red&quot;)</td>
</tr>
<tr>
<td>D # f :: Number, String -&gt; Image</td>
<td>f(5, &quot;outline&quot;) is star(5, &quot;outline&quot;, &quot;yellow&quot;)</td>
</tr>
<tr>
<td>E # f :: String, Number -&gt; Image</td>
<td>f(&quot;blue&quot;, 8) is star(8, &quot;solid&quot;, &quot;blue&quot;)</td>
</tr>
</tbody>
</table>

Match each set of examples (left) with the contract that best describes it (right).
**gt**

**Directions**: Define a function called `gt`, which makes solid green triangles of whatever size we want.

**Every contract has three parts...**

<table>
<thead>
<tr>
<th>function name</th>
<th>Number</th>
<th>-</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>domain</td>
<td></td>
<td>range</td>
</tr>
</tbody>
</table>

**Write some examples, then circle and label what changes...**

**examples:**

- Function name: `gt`, input(s) `10` is `triangle(10, "solid", "green")`
- Function name: `gt`, input(s) `20` is `triangle(20, "solid", "green")`

**Write the definition, giving variable names to all your input values...**

```haskell
fun gt( size ):
  triangle(size, "solid", "green")
```

---

**bc**

**Directions**: Define a function called `bc`, which makes solid blue circles of whatever radius we want.

**Every contract has three parts...**

<table>
<thead>
<tr>
<th>function name</th>
<th>Number</th>
<th>-</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>domain</td>
<td></td>
<td>range</td>
</tr>
</tbody>
</table>

**Write some examples, then circle and label what changes...**

**examples:**

- Function name: `bc`, input(s) `radius` is `circle(radius, "blue")`
- Function name: `bc`, input(s) `radius` is `circle(radius, "blue")`

**Write the definition, giving variable names to all your input values...**

```haskell
fun bc( radius ):
  circle(radius, "blue")
```
Solving Word Problems

Being able to see functions as Contracts, Examples or Definitions is like having three powerful tools. These representations can be used together to solve word problems!

1) When reading a word problem, the first step is to figure out the **Contract** for the function you want to build. Remember, a Contract must include the Name, Domain and Range for the function!

2) Then we write a **Purpose Statement**, which is a short note that tells us what the function **should do**. Professional programmers work hard to write good purpose statements, so that other people can understand the code they wrote!

3) Next, we write at least two **Examples**. These are lines of code that show what the function should do for a **specific** input. Once we see examples of at least two inputs, we can **find a pattern** and see which parts are changing and which parts aren’t.

4) To finish the Examples, we circle the parts that are changing, and label them with a short **variable name** that explains what they do.

5) Finally, we define the function itself! This is pretty easy after you have some examples to work from: we copy everything that didn’t change, and replace the changeable stuff with the variable name!
Creating Contracts From Examples

Write the contracts used to create each of the following collections of examples.

1)  
examples:
  big-triangle(100, "red") is
  triangle(100, "solid", "red")
  big-triangle(200, "orange") is
  triangle(200, "solid", "orange")
end

2)  
examples:
  purple-square(15) is
  rectangle(15, 15, "outline", "purple")
  purple-square(6) is
  rectangle(6, 6, "outline", "purple")
end

3)  
examples:
  banner("Game Today!") is
  text("Game Today!", 50, "red")
  banner("Go Team!") is
  text("Go Team!", 50, "red")
  banner("Exit") is
  text("Exit", 50, "red")
end

4)  
examples:
  twinkle("outline", "red") is
  star(5, "outline", "red")
  twinkle("solid", "pink") is
  star(5, "solid", "pink")
  twinkle("outline", "grey") is
  star(5, "outline", "grey")
end

5)  
examples:
  half(5) is 5 / 2
  half(8) is 8 / 2
  half(900) is 900 / 2
end
Writing Examples from Purpose Statements

We've provided contracts and purpose statements to describe two different functions. Write examples for each of those functions.

### Contract and Purpose Statement

Every contract has three parts...

**Product-squared:**

- **Function name**: `product-squared`
- **Domain**: `Number, Number`
- **Range**: `Number`

Consumes two numbers and squares their product

**Examples**

Write some examples, then circle and label what changes...

**Examples:**

- `function name (input(s))` is `what the function produces`
- `function name (input(s))` is `what the function produces`

### Contract and Purpose Statement

Every contract has three parts...

**Upside-down:**

- **Function name**: `upside-down`
- **Domain**: `Image`
- **Range**: `Image`

Consumes an image, and flips it upside down by rotating it 180 degrees.

**Examples**

Write some examples, then circle and label what changes...

**Examples:**

- `function name (input(s))` is `what the function produces`
- `function name (input(s))` is `what the function produces`

end
Directions: A rocket blasts off, and is now traveling at a constant velocity of 7 meters per second. Use the Design Recipe to write a function `rocket-height`, which takes in a number of seconds and calculates the height.

Contract and Purpose Statement

Every contract has three parts...

```
# function name :: domain -> range
#
```

Examples

Write some examples, then circle and label what changes...

**examples:**

```
function name (input/1) is what the function produces
function name (input/1) is what the function produces
end
```

Definition

Write the definition, giving variable names to all your input values...

```
fun function name (variable(s)):

what the function does with those variable(s)
end
```
Directions: Write a function double-radius, which takes in a radius and a color. It produces an outlined circle of whatever color was passed in, whose radius is twice as big as the input.

Contract and Purpose Statement
Every contract has three parts...

# function name : domain -> range
#

Examples
Write some examples, then circle and label what changes...

examples:

function name (input) is what the function produces
function name (input) is what the function produces

end

Definition
Write the definition, giving variable names to all your input values...

fun (variable(s)):

what the function does with those variable(s)

end
Word Problem: double-width

Directions: Write a function `double-width`, which takes in a number (the length of a rectangle) and produces a rectangle whose length is twice the given length.

Contract and Purpose Statement

Every contract has three parts...

<table>
<thead>
<tr>
<th>function name</th>
<th>domain</th>
<th>-&gt;</th>
<th>range</th>
</tr>
</thead>
</table>

Examples

Write some examples, then circle and label what changes...

examples:

- function name (input) is what the function produces
- function name (input) is what the function produces

Definition

Write the definition, giving variable names to all your input values...

fun (variable(s)):

what the function does with those variable(s)

end
Word Problem: next-position

Directions: Write a function next-position, which takes in two numbers (an x- and y-coordinate) and returns a DeliveryState, increasing the x-coordinate by 5 and decreasing the y-coordinate by 5.

Contract and Purpose Statement

Every contract has three parts...

# function name :: domain -> range
#
# what does the function do?

Examples

Write some examples, then circle and label what changes...

elements:

* function name (input(s)) is what the function produces

end

Definition

Write the definition, giving variable names to all your input values...

fun (variable(s)):

what the function does with those variable(s)

end
# A CakeType is a flavor, layers, & is-IceCream

data CakeType:
    | cake(
    |   ______________________________________________________
    |   ______________________________________________________
    |   ______________________________________________________
    end

to make an instance of this structure, I would write:

cake1 = ______________________________________________________
cake2 = ______________________________________________________

to access the fields of cake2, I would write:

    ______________________________________________________
    ______________________________________________________
    ______________________________________________________
    ______________________________________________________
Word Problem: taller-than

**Directions**: Write a function called `taller-than`, which consumes two CakeTypes, and produces true if the number of layers in the first CakeType is greater than the number of layers in the second.

### Contract and Purpose Statement

Every contract has three parts...

<table>
<thead>
<tr>
<th>function name</th>
<th>::</th>
<th>domain</th>
<th>-&gt;</th>
<th>range</th>
</tr>
</thead>
</table>

### Examples

Write some examples, then circle and label what changes...

**examples**:  

<table>
<thead>
<tr>
<th>function name</th>
<th>(input/arg)</th>
<th>is</th>
<th>what the function produces</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**end**

### Definition

Write the definition, giving variable names to all your input values...

**fun** (variable(s)):

<table>
<thead>
<tr>
<th>function name</th>
<th>variable(s)</th>
<th>what the function does with those variable(s)</th>
</tr>
</thead>
</table>
Directions: Write a function called will-melt, which takes in a CakeType and a temperature, and returns true if the temperature is greater than 32 degrees, AND the CakeType is an ice-cream cake.

Contract and Purpose Statement
Every contract has three parts...
#
function name : domain -> range
#
what does the function do?

Examples
Write some examples, then circle and label what changes...

eamples:

function name (input(s)) is what the function produces
function name (input(s)) is what the function produces

end

Definition
Write the definition, giving variable names to all your input values...

fun (variable(s)):

what the function does with those variable(s)

end
Below is a new structure definition:

```plaintext
data MediaType:
  | book(
    title :: String,
    author :: String,
    pubyear :: Number)
end

# an example book:
```

Fill in the blanks below with the vocabulary term that applies to each name. Here are the terms to choose from:

<table>
<thead>
<tr>
<th>contract</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>header</td>
<td>field</td>
</tr>
<tr>
<td>data type</td>
<td>instance</td>
</tr>
<tr>
<td>constructor</td>
<td>data block</td>
</tr>
<tr>
<td>name</td>
<td>purpose</td>
</tr>
</tbody>
</table>

author is a .............................................................................
book is a .............................................................................
MediaType is a .............................................................................
book1 is a .............................................................................
title is a .............................................................................
data ... end is a .............................................................................
Identifying Animation Data Worksheet

Draw a sketch for three distinct moments of the animation

Sketch A

Sketch B

Sketch C

<table>
<thead>
<tr>
<th>Thing</th>
<th>Describe how it changes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field name (dangerX, score, playerIMG ...)</th>
<th>Data Type (Number, String, Image, Boolean ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Design a Data Structure

Make a sample instance for each sketch from the previous page:

```plaintext
# a _______________State is ________________________________________
data _______________State:
  | __________( ____________________________________________________
  |
  |
  |
  |
  |
  |
  end
```

sketchA = ____________________________________________________
sketchB = ____________________________________________________
sketchC = ____________________________________________________
Write a function called `draw-state`, which takes in a `SunsetState` and returns an image in which the sun (a circle) appears at the position given in the `SunsetState`. The sun should be behind the horizon (the ground) once it is low in the sky.

```plaintext
draw-state :: _______________ -> Image
# __________________________________________________________________

<table>
<thead>
<tr>
<th>SUN =</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROUND =</td>
</tr>
<tr>
<td>SKY =</td>
</tr>
</tbody>
</table>

fun __________ (____________________________________):  

```
# __________________________________________________________________
```  

end
```
Word Problem: next-state-tick

Directions: Write a function called `next-state-tick`, which takes in a SunsetState and returns a SunsetState in which the new x-coordinate is 8 pixels larger than in the given SunsetState and the y-coordinate is 4 pixels smaller than in the given SunsetState.

Contract and Purpose Statement

Every contract has three parts...

<table>
<thead>
<tr>
<th>function name</th>
<th>domain</th>
<th>range</th>
</tr>
</thead>
</table>

Examples

Write some examples, then circle and label what changes...

examples:

<table>
<thead>
<tr>
<th>function name</th>
<th>input(s)</th>
<th>what the function produces</th>
</tr>
</thead>
</table>

Definition

Write the definition, giving variable names to all your input values...

fun (variable(s)):

what the function does with those variable(s)

end
# Identifying Animation Data Worksheet

<table>
<thead>
<tr>
<th>Sketch A</th>
<th>Sketch B</th>
<th>Sketch C</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Thing</strong></th>
<th><strong>Describe how it changes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Field name (dangerX, score, playerIMG ...)</strong></th>
<th><strong>Data Type (Number, String, Image, Boolean ...)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Design a Data Structure

# a _______________ State is ______________________________________
data _______________ State:
  |
  | ______________________________________________________________
  | ______________________________________________________________
  | ______________________________________________________________

end

Make a sample instance for each sketch from the previous page:

    sketchA  = ____________________________________________________

    sketchB  = ____________________________________________________

    sketchC  = ____________________________________________________
## Identifying Animation Data Worksheet

<table>
<thead>
<tr>
<th>Sketch A</th>
<th>Sketch B</th>
<th>Sketch C</th>
</tr>
</thead>
</table>

### What things are changing?

<table>
<thead>
<tr>
<th>Thing</th>
<th>Describe how it changes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Field name (dangerX, score, playerIMG …) | Data Type (Number, String, Image, Boolean …)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

50
Design a Data Structure

Make a sample instance for each sketch from the previous page:

```plaintext
# a ___________________ State is ________________________________

data __________________ State:
  | _____________________ ____________________________
                                          ____________________________
                                          ____________________________
                                          ____________________________
end

Make a sample instance for each sketch from the previous page:

```
### Identifying Animation Data Worksheet

<table>
<thead>
<tr>
<th>Sketch A</th>
<th>Sketch B</th>
<th>Sketch C</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Thing</th>
<th>Describe how it changes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field name (dangerX, score, playerIMG ...)</th>
<th>Data Type (Number, String, Image, Boolean ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Design a Data Structure

Make a sample instance for each sketch from the previous page:

```
# a ________________ State is ________________________________

data ________________ State:
  | __________{
  |
  | |
  | |
  | |
  | end

Make a sample instance for each sketch from the previous page:

sketchA = ____________________________________________

sketchB = ____________________________________________

sketchC = ____________________________________________
```
Identifying Animation Data Worksheet

Draw a sketch for three distinct moments of the animation

Sketch A  Sketch B  Sketch C

<table>
<thead>
<tr>
<th>Thing</th>
<th>Describe how it changes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field name (dangerX, score, playerIMG ...)</th>
<th>Data Type (Number, String, Image, Boolean ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Design a Data Structure

Make a sample instance for each sketch from the previous page:

```plaintext
sketchA = 
```

```plaintext
sketchB = 
```

```plaintext
sketchC = 
```
**Word Problem: location**

**Directions**: Write a function called `location`, which consumes a `DeliveryState`, and produces a `String` representing the location of a box: either “road”, “delivery zone”, “house”, or “air”.

**Contract and Purpose Statement**

Every contract has three parts...

```
# function name :: domain -> range
#
```

**Examples**

Write some examples, then circle and label what changes...

**examples:**

```
function name (input) is what the function produces
function name (input) is what the function produces
function name (input) is what the function produces
function name (input) is what the function produces
```

**Definition**

Write the definition, giving variable names to all your input values...

```
fun function name (variable(s)):

what the function does with those variable(s)
```

end
<table>
<thead>
<tr>
<th>Buggy Code</th>
<th>Correct Code / Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong></td>
<td><code>fun piecewisefun(n):</code>&lt;br&gt;<code>if (n &gt; 0): n</code>&lt;br&gt;<code>else: 0</code></td>
</tr>
<tr>
<td><strong>2</strong></td>
<td><code>fun cost(topping):</code>&lt;br&gt;<code>if string-equal(topping, &quot;pepperoni&quot;): 10.50</code>&lt;br&gt;<code>else string-equal(topping, &quot;cheese&quot;): 9.00</code>&lt;br&gt;<code>else string-equal(topping, &quot;chicken&quot;): 11.25</code>&lt;br&gt;<code>else string-equal(topping, &quot;broccoli&quot;): 10.25</code>&lt;br&gt;<code>else: &quot;That's not on the menu!&quot;</code>&lt;br&gt;<code>end end</code></td>
</tr>
<tr>
<td><strong>3</strong></td>
<td><code>fun absolute-value(a b):</code>&lt;br&gt;<code>if a &gt; b: a - b</code>&lt;br&gt;<code>b - a</code>&lt;br&gt;<code>end end</code></td>
</tr>
<tr>
<td><strong>4</strong></td>
<td><code>fun best-function(f):</code>&lt;br&gt;<code>if string-equal(f, &quot;blue&quot;):</code>&lt;br&gt;&quot;you win!&quot;&lt;br&gt;<code>else if string-equal(f, &quot;blue&quot;):</code>&lt;br&gt;&quot;you lose!&quot;&lt;br&gt;<code>else if string-equal(f, &quot;red&quot;):</code>&lt;br&gt;&quot;Try again!&quot;&lt;br&gt;<code>else: &quot;Invalid entry!&quot;</code>&lt;br&gt;<code>end end</code></td>
</tr>
</tbody>
</table>
Decrease the cat's hunger level by 2 and sleep level by 1 on each tick.

<table>
<thead>
<tr>
<th>Sketch A</th>
<th>Sketch B</th>
<th>Sketch C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thing</th>
<th>Describe how it changes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field name (dangerX, score, playerIMG ...)</th>
<th>data type (Number, String, Image, Boolean ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Component</th>
<th>When is there work to be done?</th>
<th>To-Do</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Structure</td>
<td>If any new field(s) were added, changed, or removed</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>draw-state</td>
<td>If something is displayed in a new way or position</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>next-state-tick</td>
<td>If the Data Structure changed, or the animation happens automatically</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>next-state-key</td>
<td>If the Data Structure changed, or a keypress triggers the animation</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>reactor</td>
<td>If either next-state function is new</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>
1) Make a sample instance for each sketch from the previous page:

\[
\text{function1} = \ldots
\]

\[
\text{function2} = \ldots
\]

\[
\text{function3} = \ldots
\]

\[
\text{function4} = \ldots
\]

2) Write at least one NEW example for one of the functions on your To-Do list

\[
\text{new_example1} = \ldots
\]

\[
\text{new_example2} = \ldots
\]

\[
\text{new_example3} = \ldots
\]

3) If you have another function on your To-Do list, write at least one NEW example

\[
\text{new_example4} = \ldots
\]

\[
\text{new_example5} = \ldots
\]
Word Problem: draw-sun

Directions: Write a function called draw-sun, which consumes a SunsetState, and produces an image of a sun (a solid, 25 pixel circle), whose color is "yellow", when the sun's y-coordinate is greater than 225, "orange", when its y-coordinate is between 150 and 225, and "red" otherwise.

Contract and Purpose Statement

Every contract has three parts...

# function name :: domain -> range
#
# what does the function do?

Examples

Write some examples, then circle and label what changes...

examples:

( ) is
( ) is
( ) is
( ) is

Definition

Write the definition, giving variable names to all your input values...

fun ( ) :

():

what the function does with those variable(s)
Decrease the cat’s hunger level by 2 and sleep level by 1 on each tick.

<table>
<thead>
<tr>
<th>Sketch A</th>
<th>Sketch B</th>
<th>Sketch C</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Thing</th>
<th>Describe how it changes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field name (dangerX, score, playerIMG ...)</th>
<th>data type (Number, String, Image, Boolean ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Component</th>
<th>When is there work to be done?</th>
<th>To-Do</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Structure</td>
<td>If any new field(s) were added, changed, or removed</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>draw-state</td>
<td>If something is displayed in a new way or position</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>next-state-tick</td>
<td>If the Data Structure changed, or the animation happens automatically</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>next-state-key</td>
<td>If the Data Structure changed, or a keypress triggers the animation</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>reactor</td>
<td>If either next-state function is new</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>
1) Make a sample instance for each sketch from the previous page:

\[
\begin{align*}
\text{FULLPET} & = \text{pet}(100, 100) \\
\text{MIDPET} & = \text{pet}(50, 75) \\
\text{LOSEPET} & = \text{pet}(0, 0)
\end{align*}
\]

2) Write at least one NEW example for one of the functions on your To-Do list

\[
\begin{align*}
\text{next-state-tick(FULLPET)} & \text{ is pet(FULLPET.hunger – 2, FULLPET.sleep – 1)} \\
\text{next-state-tick(MIDPET)} & \text{ is pet(MIDPET.hunger – 2, MIDPET.sleep – 1)} \\
\text{next-state-tick(LOSEPET)} & \text{ is LOSEPET}
\end{align*}
\]

3) If you have another function on your To-Do list, write at least one NEW example
Decrease the cat’s hunger level by 2 and sleep level by 1 on each tick.

<table>
<thead>
<tr>
<th>Sketch A</th>
<th>Sketch B</th>
<th>Sketch C</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Thing</th>
<th>Describe how it changes</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Field name (dangerX, score, playerIMG ...)</th>
<th>data type (Number, String, Image, Boolean ...)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Component</th>
<th>When is there work to be done?</th>
<th>To-Do</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Structure</td>
<td>If any new field(s) were added, changed, or removed</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>draw-state</td>
<td>If something is displayed in a new way or position</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>next-state-tick</td>
<td>If the Data Structure changed, or the animation happens automatically</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>next-state-key</td>
<td>If the Data Structure changed, or a keypress triggers the animation</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>reactor</td>
<td>If either next-state function is new</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>
1) Make a sample instance for each sketch from the previous page:

\[ \_ = \]  

\[ \_ = \]  

\[ \_ = \]  

\[ \_ = \]  

2) Write at least one NEW example for one of the functions on your To-Do list

\[ \_ = \]  

\[ \_ = \]  

\[ \_ = \]  

\[ \_ = \]  

\[ \_ = \]  

\[ \_ = \]  

3) If you have another function on your To-Do list, write at least one NEW example

\[ \_ = \]  

\[ \_ = \]  

\[ \_ = \]  

\[ \_ = \]  

\[ \_ = \]  

\[ \_ = \]
Decrease the cat’s hunger level by 2 and sleep level by 1 on each tick.

<table>
<thead>
<tr>
<th>Sketch A</th>
<th>Sketch B</th>
<th>Sketch C</th>
</tr>
</thead>
</table>

**Thing** | **Describe how it changes**
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Field name (dangerX, score, playerIMG ...)** | **data type (Number, String, Image, Boolean ...)**
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Component</strong></th>
<th><strong>When is there work to be done?</strong></th>
<th><strong>To-Do</strong></th>
<th><strong>Done</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Structure</td>
<td><em>If any new field(s) were added, changed, or removed</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>draw-state</td>
<td><em>If something is displayed in a new way or position</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>next-state-tick</td>
<td><em>If the Data Structure changed, or the animation happens automatically</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>next-state-key</td>
<td><em>If the Data Structure changed, or a keypress triggers the animation</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reactor</td>
<td><em>If either next-state function is new</em></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1) Make a sample instance for each sketch from the previous page:

\[
\text{________} = \\
\text{__________________________________________}
\]

\[
\text{________} = \\
\text{__________________________________________}
\]

\[
\text{________} = \\
\text{__________________________________________}
\]

\[
\text{________} = \\
\text{__________________________________________}
\]

2) Write at least one NEW example for one of the functions on your To-Do list

\[
\text{------------------------------------------------------------------}
\]

\[
\text{------------------------------------------------------------------}
\]

\[
\text{------------------------------------------------------------------}
\]

\[
\text{------------------------------------------------------------------}
\]

3) If you have another function on your To-Do list, write at least one NEW example

\[
\text{------------------------------------------------------------------}
\]

\[
\text{------------------------------------------------------------------}
\]

\[
\text{------------------------------------------------------------------}
\]

\[
\text{------------------------------------------------------------------}
\]
Decrease the cat’s hunger level by 2 and sleep level by 1 on each tick.

<table>
<thead>
<tr>
<th>Sketch A</th>
<th>Sketch B</th>
<th>Sketch C</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Thing</th>
<th>Describe how it changes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field name (dangerX, score, playerIMG ...)</th>
<th>data type (Number, String, Image, Boolean ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Component</th>
<th>When is there work to be done?</th>
<th>To-Do</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Structure</td>
<td>If any new field(s) were added, changed, or removed</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>draw-state</td>
<td>If something is displayed in a new way or position</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>next-state-tick</td>
<td>If the Data Structure changed, or the animation happens automatically</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>next-state-key</td>
<td>If the Data Structure changed, or a keypress triggers the animation</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>reactor</td>
<td>If either next-state function is new</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>
Define the Data Structure

# a
State is

data

|   |
|   | (  
|   |   
|   |   
|   |   
|
end

Make a sample instance for each sketch from the previous page

= 
= 
= 

Write an example for one of the functions on the previous page
Distance

The Player is at (4, 2) and the Target is at (0, 5).
Distance takes in the player’s x, player’s y, character’s x and character’s y. Use the formula below to fill in the EXAMPLE:
\[ \sqrt{(4 - 0)^2 + (2 - 5)^2} \]

Convert it into a Circle of Evaluation. (We've already gotten you started!)

Convert it to Pyret code.
Word Problem: distance

Directions: Write a function `distance`, which takes FOUR inputs: (1) px: The x-coordinate of the player, (2) py: The y-coordinate of the player, (3) cx: The x-coordinate of another game character, (4) cy: The y-coordinate of another game character. It should return the distance between the two, using the Distance formula: \( \text{Distance}^2 = (px - cx)^2 + (py - cy)^2 \)

Contract and Purpose Statement

Every contract has three parts...

```
# function name :: domain -> range
#
# what does the function do?
```

Examples

Write some examples, then circle and label what changes...

examples:

```
function name (input(s)) is what the function produces
function name (input(s)) is what the function produces
```

Definition

Write the definition, giving variable names to all your input values...

```
fun (variable(s)) :
  function name variable(s)
  what the function does with those variable(s)
end
```
Directions: Write a function `is-collision`, which takes FOUR inputs: (1) `px`: The x-coordinate of the player, (2) `py`: The y-coordinate of the player, (3) `cx`: The x-coordinate of another game character, (4) `cy`: The y-coordinate of another game character. It should return `true` if the coordinates of the player are within 50 pixels of the coordinates of the other character. Otherwise, `false`.

**Contract and Purpose Statement**

Every contract has three parts...

```plaintext
function name :: domain -> range
#
# what does the function do?
#
```

**Examples**

Write some examples, then circle and label what changes...

```plaintext
examples:

  function name (input(s)) is what the function produces
  function name (input(s)) is what the function produces

end
```

**Definition**

Write the definition, giving variable names to all your input values...

```plaintext
fun (variable(s)):
  function name (variable(s))
  what the function does with those variable(s)

end
```
Directions:

**Contract and Purpose Statement**
Every contract has three parts...

<table>
<thead>
<tr>
<th>function name</th>
<th>domain</th>
<th>-&gt;</th>
<th>range</th>
</tr>
</thead>
</table>

Examples
Write some examples, then circle and label what changes...

**Examples:**

- function name (input(s)) is what the function produces
- function name (input(s)) is what the function produces

**Definition**
Write the definition, giving variable names to all your input values...

**fun** (variable(s)):

function name variable(s)

what the function does with those variable(s)

end
Directions:

Contract and Purpose Statement
Every contract has three parts...

<table>
<thead>
<tr>
<th>#</th>
<th>function name</th>
<th>::</th>
<th>domain</th>
<th>-&gt;</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>what does the function do?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Examples
Write some examples, then circle and label what changes...

eamples:

<table>
<thead>
<tr>
<th>function name</th>
<th>input(s)</th>
<th>is</th>
<th>what the function produces</th>
</tr>
</thead>
<tbody>
<tr>
<td>function name</td>
<td>input(s)</td>
<td>is</td>
<td>what the function produces</td>
</tr>
</tbody>
</table>

end

Definition
Write the definition, giving variable names to all your input values...

fun | variable(s) |
| function name | variable(s) |

end
Decrease the cat’s hunger level by 2 and sleep level by 1 on each tick.

**Sketch A**

**Sketch B**

**Sketch C**

<table>
<thead>
<tr>
<th>Thing</th>
<th>Describe how it changes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field name (dangerX, score, playerIMG ...)</th>
<th>Datatype (Number, String, Image, Boolean ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Component</th>
<th>When is there work to be done?</th>
<th>To-Do</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Structure</td>
<td>If any new field(s) were added, changed, or removed</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>draw-state</td>
<td>If something is displayed in a new way or position</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>next-state-tick</td>
<td>If the Data Structure changed, or the animation happens automatically</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>next-state-key</td>
<td>If the Data Structure changed, or a keypress triggers the animation</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>reactor</td>
<td>If either next-state function is new</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>
# a  
State is  

data  
|   (  
   
   
   )
end

=  
=  
=  

Write an example for one of the functions on the previous page
Animation Data Worksheet

Decrease the cat’s hunger level by 2 and sleep level by 1 on each tick.

<table>
<thead>
<tr>
<th>Sketch A</th>
<th>Sketch B</th>
<th>Sketch C</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Thing</th>
<th>Describe how it changes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field name (dangerX, score, playerIMG ...)</th>
<th>Datatype (Number, String, Image, Boolean ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Component</th>
<th>When is there work to be done?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Structure</td>
<td>If any new field(s) were added, changed, or removed</td>
</tr>
<tr>
<td>draw-state</td>
<td>If something is displayed in a new way or position</td>
</tr>
<tr>
<td>next-state-tick</td>
<td>If the Data Structure changed, or the animation happens automatically</td>
</tr>
<tr>
<td>next-state-key</td>
<td>If the Data Structure changed, or a keypress triggers the animation</td>
</tr>
<tr>
<td>reactor</td>
<td>If either next-state function is new</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>To-Do</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>
Define the Data Structure

# a State is ........................................
data _______ State:
| _______ ( ........................................

 ........................................
end

= 
= 
= 

Write an example for one of the functions on the previous page
Decrease the cat’s hunger level by 2 and sleep level by 1 on each tick.

<table>
<thead>
<tr>
<th>Sketch A</th>
<th>Sketch B</th>
<th>Sketch C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thing</th>
<th>Describe how it changes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field name (dangerX, score, playerIMG ...)</th>
<th>Datatype (Number, String, Image, Boolean ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Component</th>
<th>When is there work to be done?</th>
<th>To-Do</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Structure</td>
<td>If any new field(s) were added, changed, or removed</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>draw-state</td>
<td>If something is displayed in a new way or position</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>next-state-tick</td>
<td>If the Data Structure changed, or the animation happens automatically</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>next-state-key</td>
<td>If the Data Structure changed, or a keypress triggers the animation</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>reactor</td>
<td>If either next-state function is new</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>
Define the Data Structure

# a  State is  
data  State:
|   (  
   
   
   )
end

Make a sample instance for each sketch from the previous page
  =  
  =  
  =  

Write an example for one of the functions on the previous page
Contracts tell us how to use a function. For example:

`num-sqr :: (n :: Number) -> Number` tells us that the name of the function is `num-sqr`, it takes one input (a `Number`), and it evaluates to a `Number`. From the contract, we know `num-sqr(4)` will evaluate to a `Number`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Contracts tell us how to use a function. For example: `num-sqr :: (n :: Number) -> Number` tells us that the name of the function is `num-sqr`, it takes one input (a `Number`), and it evaluates to a `Number`. From the contract, we know `num-sqr(4)` will evaluate to a `Number`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td># above</td>
<td>$(\text{img1} :: \text{Image}, \text{img2} :: \text{Image})$</td>
<td>$\rightarrow$ $\text{Image}$</td>
</tr>
<tr>
<td># put-image</td>
<td>$(\text{img1} :: \text{Image}, x :: \text{Number}, y :: \text{Number}, \text{img2} :: \text{Image})$</td>
<td>$\rightarrow$ $\text{Image}$</td>
</tr>
<tr>
<td># rotate</td>
<td>$(\text{degree} :: \text{Number}, \text{img} :: \text{Image})$</td>
<td>$\rightarrow$ $\text{Image}$</td>
</tr>
<tr>
<td># scale</td>
<td>$(\text{factor} :: \text{Number}, \text{img} :: \text{Image})$</td>
<td>$\rightarrow$ $\text{Image}$</td>
</tr>
<tr>
<td># string-repeat</td>
<td>$(\text{text} :: \text{String}, \text{repeat} :: \text{Number})$</td>
<td>$\rightarrow$ $\text{String}$</td>
</tr>
<tr>
<td># string-contains</td>
<td>$(\text{text} :: \text{String}, \text{search-for} :: \text{String})$</td>
<td>$\rightarrow$ $\text{Boolean}$</td>
</tr>
<tr>
<td># num-sqr</td>
<td>$(n :: \text{Number})$</td>
<td>$\rightarrow$ $\text{Number}$</td>
</tr>
<tr>
<td># num-sqrt</td>
<td>$(n :: \text{Number})$</td>
<td>$\rightarrow$ $\text{Number}$</td>
</tr>
<tr>
<td># num-min</td>
<td>$(\text{a} :: \text{Number}, \text{b} :: \text{Number})$</td>
<td>$\rightarrow$ $\text{Number}$</td>
</tr>
<tr>
<td># num-max</td>
<td>$(\text{a} :: \text{Number}, \text{b} :: \text{Number})$</td>
<td>$\rightarrow$ $\text{Number}$</td>
</tr>
</tbody>
</table>
Contracts tell us how to use a function. For example:

```
num-sqr :: (n :: Number) -> Number
```

tells us that the name of the function is `num-sqr`, it takes one input (a `Number`), and it evaluates to a `Number`.

From the contract, we know `num-sqr(4)` will evaluate to a `Number`.

### Domain

- `num-sqr` takes a single input of type `Number`.
- It returns a single output of type `Number`.

### Name

- `num-sqr` is the name of the function.

### Range

- The range is a `Number`.

### String-equal

```
string-equal :: (str1 :: String, str2 :: String) -> Boolean
```

### Or

```
or :: (test1 :: Boolean, test2 :: Boolean) -> Boolean
```

### And

```
and :: (test1 :: Boolean, test2 :: Boolean) -> Boolean
```

### Not

```
ot :: ()
```

### Arithmetic Operations

- `+`, `-`, `*`, `/`
- `+ :: (a :: Number, b :: Number) -> Number`
- `- :: (a :: Number, b :: Number) -> Number`
- `* :: (a :: Number, b :: Number) -> Number`
- `/ :: (a :: Number, b :: Number) -> Number`

### Comparison Operations

- `==`, `!=`, `<`, `>`, `<=`, `>=`
- `== :: (a :: String, b :: String) -> Boolean`
- `!= :: (a :: String, b :: String) -> Boolean`
- `< :: (a :: Number, b :: Number) -> Boolean`
- `>` :: (a :: Number, b :: Number) -> Boolean`
- `<= :: (a :: Number, b :: Number) -> Boolean`
- `>= :: (a :: Number, b :: Number) -> Boolean`

### Logical Operations

- `&&`, `||`
- `&& :: (test1 :: Boolean, test2 :: Boolean) -> Boolean`
- `|| :: (test1 :: Boolean, test2 :: Boolean) -> Boolean`

### Relational Operations

- `>`, `<`, `>=`, `<=`
- `>` :: (a :: Number, b :: Number) -> Boolean`
- `< :: (a :: Number, b :: Number) -> Boolean`
- `>= :: (a :: Number, b :: Number) -> Boolean`
- `<= :: (a :: Number, b :: Number) -> Boolean`
These materials were developed partly through support of the National Science Foundation, (awards 1042210, 1535276, 1648684, and 1738598), and are licensed under a Creative Commons 4.0 Unported License. Based on a work at www.BootstrapWorld.org. Permissions beyond the scope of this license may be available by contacting schanzer@BootstrapWorld.org.