Sam the Butterfly - Applying Inequalities

(Also available in WeScheme)

Students discover that inequalities have an important application in video games: keeping game characters on the screen! Students apply their understanding to edit code so that it will keep Sam the Butterfly safely in view.

Lesson Goals	Students will be able to: • apply their understanding of inequalities to keep a game character on the screen
Student-Facing Lesson Goals	• Let's use what we know about inequalities to define the boundaries that will keep a game character on the screen.
Prerequisites	• Simple Data Types • Contracts • Simple Inequalities • Compound Inequalities: Solutions & Non-Solutions
Materials	• PDF of all Handouts and Page • Sam the Butterfly Starter File • Lesson Slides • Printable Lesson Plan (a PDF of this web page)
Supplemental Materials	• Additional Printable Pages for Scaffolding and Practice

Glossary

inequality :: a mathematical description of the relationship between two variables or quantities, in which they are not necessarily equal

Introducing Sam *15 minutes*

Overview

Students are introduced to Sam the Butterfly, a simple activity in which they must write simple ***inequalities*** to detect when Sam has gone too far in one dimension.

Launch

- Open the Sam the Butterfly Starter File in a new tab and save a copy of your own.
- Complete Introducing Sam, clicking "Run" and using the arrow keys to investigate the program with your partner.

As students explore the program, they should notice that Sam's coordinates are displayed at the top of the screen. When Sam is at (0,0), we only see a part of Sam's wing because Sam's position is based on the *center* of the butterfly image. Students should observe that Sam can go up to, but not beyond, an x of -50. Students can represent this algebraically as $x > -50$, or (for students who notice that Sam only moves in increments of 10) $x \geq -40$.

> Every time Sam moves, we want to check and see if Sam is safe.

To further support students, consider asking what three functions are defined in their starter files. Then, ask students what each function *should* do, when working properly.

- What *should* our left-checking function do?
  - *Check to see if x is greater than -50.*
- What *should* our right-checking function do?
  - *Check to see if x is less than 690.*
- What should `is-onscreen(x, y)` do?
  - *Answers may vary. Let students drive the discussion, and don't give away the answer!*

Why does `is-onscreen(x, y)` take in both x and y?

A common misconception about functions is that they need to use all of their variables. This misconception shows up as soon as students see a horizontal line written using function notation: $f(x) = 15$ doesn't use the variable x at all. Function definitions can also take in many variables and only use some of them, for instance $f(g,m) = 15g$.

In this case, the "screen" that `is-onscreen(x, y)` refers to is two-dimensional and Sam moves in both directions. Even though `is-onscreen` is tracking both Sam's x- and y-coordinates, for now we're looking to write a function that only evaluates where Sam is on the **left** and **right**. In other words, `is-onscreen(x, y)` takes in two variables, but *only uses one of them!*

When we add this function to our Game Starter File, it's important that `is-onscreen(x, y)` only changes the x-coordinate, because the Game Starter File has other (hidden) functions that will control the characters' y-coordinates and we don't want to interfere with them.

However, later on in *this lesson* we will discuss how students with time and energy can challenge themselves to keep Sam safe in all directions by defining a function that uses both of the variables it takes in!

Investigate

- Complete Left and Right with your partner.
- Once finished, fix the corresponding functions in your Sam the Butterfly file, and test them out.

Students will notice that fixing `is-safe-left` keeps Sam from disappearing off the left side, but fixing `is-safe-right` doesn't seem to keep Sam from disappearing off the right side! When students encounter this, encourage them to look through the code to try and figure out why.

"False" doesn't mean "Wrong"!

A lot of students - especially confident ones - may struggle to come up with an example where `is-safe-left` returns `false`:

```
# Students hate writing the second one!
examples:
is-safe-left(189) is 189 > -50
is-safe-left(-65) is -65 > -50
end
```

This misconception comes from confusing a statement that is "false" with a program that is "wrong". In the second example, above, the result of `is-safe-left(-65)` *is* `false`, because "65 is greater than -50" *is a false statement*. Remind your students that you want one example that's true, and a second that's false!

Pyret includes some functionality that makes this more explicit, and can help resolve the misconception:

```
examples:
  is-safe-left( 89) is true  because  89 > -50
  is-safe-left(-65) is false because -65 > -50
end
```

By writing the answer first (`is-safe-left(-65) is false`), it reduces anxiety about code being "wrong". Students can think of the `because` as *an explanation of why the answer is false*.

Emphasize to students that they cannot trust the behavior of a complex system! After looking closely at examples and observing that they all pass, students should suspect that the bug is elsewhere.

Synthesize

- Does `is-safe-left` work correctly? How do you know?
- Does `is-safe-right` work correctly? How do you know?

Protecting Sam on Both Sides *30 minutes*

Overview

Students solve a word problem involving compound inequalities, using `and` to compose the simpler Boundary-checking functions from the previous lesson.

Launch

Recruit three student volunteers to roleplay the functions `is-safe-left`, `is-safe-right`, and `is-onscreen`. Give them 1 minute to read the Contract and code, as written in the program.

Ask the volunteers what their name, Domain and Range are. Explain that you, the facilitator, will be providing a coordinate input. The functions `is-safe-left` and `is-safe-right` will respond with either "true" or "false".

The function `is-onscreen`, however, will call the `is-safe-left` function! So the student roleplaying `is-onscreen` should turn to `is-safe-left` and give the input to them.

For example:

- Facilitator: "is-onscreen 70"
- is-onscreen (turns to is-safe-left): "is-safe-left 70"
- is-safe-left: "true"
- is-onscreen (turns back to facilitator): "true"

- Facilitator: "is-onscreen -100"
- is-onscreen (turns to is-safe-left): "is-safe-left -100"
- is-safe-left: "false"
- is-onscreen (turns back to facilitator): "false"

- Facilitator: "is-onscreen 900"
- is-onscreen (turns to is-safe-left): "is-safe-left 900"
- is-safe-left: "true"
- is-onscreen (turns back to facilitator): "true"

Hopefully your students will notice that `is-safe-right` did not participate in this roleplay scenario at all!

- What is the problem with `is-onscreen`?
  - *It's only talking to* `is-safe-left`, *it's not checking with* `is-safe-right`
- What should `is-onscreen` be doing?
  - *It needs to talk to* `is-safe-left` *AND* `is-safe-right`

Investigate

- Complete Word Problem: is-onscreen.
- When this function is entered into the editor, students should now see that Sam is protected on *both* sides of the screen.

Extension Option

What if we wanted to keep Sam safe on the top and bottom edges of the screen as well? What additional functions would we need? What functions would need to change? *We recommend that students tackling this challenge define a new function* `is-onscreen-2` *because they will need their original* `is-onscreen` *code in the next section of this lesson.*

Synthesize

Bring back the three new student volunteers to roleplay those functions, with the onscreen function now working properly. Make sure students provide correct answers, testing both `true` and `false` conditions using coordinates where Sam is onscreen and offscreen.

- How did it feel when you saw Sam hit both walls?
- Are there multiple solutions for `is-onscreen`?
- Is this *Top-Down* or *Bottom-Up* design?

Boundary Detection in the Game *10 minutes*

Overview

Students identify common patterns between two-dimensional Boundary detection and detecting whether a player is onscreen. They apply the same problem-solving and narrow mathematical concept from the previous lesson to a more general problem.

Launch

Have students open their in-progress game file and click "Run". Invite them to analyze the movement of the danger and the target

- How are the `TARGET` and `DANGER` behaving right now?
  - *They move across the screen.*
- What do we want to change?
  - *We want them to come back after they leave one side of the screen.*
- What happens to an image's x-coordinate when it moves off the screen?
  - *An image is entirely off-screen if its x-coordinate is less than -50 and greater than 690.*
- How can we make the computer understand when an image has moved off the screen?
  - *We can teach the computer to compare the image's coordinates to a boundary on the number line, just like we did with Sam the Butterfly!*

Investigate

Apply what you learned from Sam the Butterly to fix the `is-safe-left`, `is-safe-right`, and `is-onscreen` functions in your own code.

Since the screen dimensions for their game are 640x480, just like Sam, they can use their code from Sam as a starting point.

NOTE Students should NOT add `is-safe-top` and `is-safe-bottom` to their game code!

Common Misconceptions

- Students will need to test their code with their images to see if the boundaries are correct for them. Students with large images may need to use slightly wider boundaries, or vice versa for small images. In some cases, students may have to go back and rescale their images if they are too large or too small for the game.
- Students may be surprised that the same code that "traps Sam" also "resets the `DANGER` and `TARGET` ". It's critical to explain that these functions do *neither* of those things! All they do is test if a coordinate is within a certain range on the x-axis. There is other code (hidden in the teachpack) that determines *what to do if the coordinate is offscreen*. The ability to re-use function is one of the most powerful features of mathematics - and programming!

Synthesize

- The same code that "trapped" Sam also "resets" the `DANGER` and the `TARGET`. What is actually going on?

Additional Exercises

- Onscreen - More than One Way
- Keeping NinjaCat in the Game