Sam the Butterfly - Applying Inequalities

(Also available for 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	• I can use what I know about inequalities to define the boundaries that will keep a character on the screen.
Materials	• Lesson Slides • Introducing Sam (Page 71) • Left and Right (Page 72) • Word Problem: is-onscreen (Page 73) • Sam The Butterfly starter file (Pyret) • *Optional: Keeping NinjaCat in the Game*
Preparation	• Make sure all materials have been gathered • Decide how students will be grouped in pairs
Language Table	(see below)

Language Table

Types	Functions	Values
Number	`+` , `-` , `*` , `/` , `num-expt` , `num-sqr` , `num-sqrt`	`4` , `-1.2` , `2/3` , `pi`
String	`string-length` , `string-repeat` , `string-contains`	`"hello"` , `"91"`
Boolean	`<` , `>` , `<>` , `<=` , `>=` , `string-equal` , `<` , `>` , `==` , `<>` , `>=` , **`and`** , **`or`**	`true` , `false`
Image	`star` , `triangle` , `circle` , `square` , `rectangle` , `rhombus` , `ellipse` , `regular-polygon` , `radial-star` , `text` , `overlay` , `above` , `beside` , `rotate` , `scale` , `flip-horizontal` , `flip-vertical`	

Glossary

coordinate :: a number or set of numbers describing an object's location

expression :: a computation written in the rules of some language (such as arithmetic, code, or a Circle of Evaluation)

Introducing Sam 30 minutes

Overview

Students are introduced to Sam the Butterfly: a simple activity in which they must write 1-step inequalities to detect when Sam has gone too far in one dimension.

Launch

Have students open the
Sam The Butterfly starter file (Pyret) and click "Save A Copy."
Have students turn to the Introducing Sam (Page 71), click run and use the arrow keys to investigate the program with their partner.

- **What is something you noticed about this program?** *As Sam moves, the* ***coordinates*** *are displayed at the top of the screen; the coordinates are all in the 1st quadrant; etc.*
- **What do you see when Sam is at (0,0)? Why is that?** *You only see part of Sam's wing. Sam's position is based on the center of Sam's image.*
- **How far can Sam go to the left and stay on the screen?** *Up to, but not beyond, an x of -40.*
- **How could we write this as an *expression*?** $x \geq -40$, or $x > -50$

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

- There are three functions defined in this file. What are they?

Optional: For extra scaffolding…

- **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 `onscreen?` do?** *Answers may vary, let students drive the discussion, and don't give away the answer*

Investigate

With their partners, students complete Left and Right (Page 72). Once finished, students can fix the corresponding functions in their Sam the Butterly file, and test them out.

"False" doesn't mean "Wrong"!

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

```
# Students hate writing the second one!
examples:
is-safe-left( 89) is 89 > -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*.

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*.

Students will notice that fixing `is-safe-left` keeps Sam from disappearing off the left, 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. The answer will be revealed in the next lesson.

- Recruit three new student volunteers to roleplay those same functions, which have now been *corrected*. Make sure students provide correct answers, testing both `true` and `false` conditions using coordinates where Sam is onscreen and offscreen.

Common Misconceptions

- Many students - especially traditionally high-achieving ones - will be very concerned about writing examples that are "wrong." The misconception here is that an expression that produces `false` is somehow *incorrect*. You can preempt this in advance, by explaining that our Boolean-producing functions *should sometimes return false*, such as when Sam is offscreen.
- Push students to think carefully about corner-cases, such as when Sam is *exactly* at -50 or 690.

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 `safe-left?`, `safe-right?` and `onscreen?`. Give them 1 minute to read the contract and code, as written in the program.
- As in the previous lesson, ask the volunteers what their name, Domain and Range are, and then test them out by calling out their name, followed by a number. (For example, "(safe-left? 20)!", "(safe-right? -100)!", "(onscreen? 829)!") **Note"** the code for `onscreen` *calls the safe-left function!*. So the student roleplaying `onscreen` should turn to `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**: "onscreen-huh -100"
- **is-onscreen** (turns to is-safe-left): "safe-left-huh -100"
- **is-safe-left**: "false"
- **is-onscreen** (turns back to facilitator): "false"

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

Ask the rest of the class

- What is the problem with `is-onscreen`?
  It's only talking to `is-safe-left`*, it's not checking with* `is-safe-right`
- How can `is-onscreen` check with both?
  It needs to talk to `is-safe-left` *AND* `is-safe-right`

Have students complete Word Problem: is-onscreen (Page 73). When this functions is entered into WeScheme, 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` *to tackle this challenge.*

Boundary Detection in the Game 10 minutes

Overview

Students identify common patterns between 2-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 press Run.

- 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.

- How do we know when an image has moved off the screen?

  We can see it.

- 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

Students apply what they learned from Sam the Butterly to fix the `is-safe-left` , `is-safe-right` , and `is-onscreen` functions in their 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: Those Students who figured out the challenge of how to keep Sam safe on top and bottom, should not use the additional code they wrote.

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!

Additional Exercises

- Keeping NinjaCat in the Game