Unit 7
Conditional Branching

Unit Overview

Students use geometry and knowledge of basic image functions to design characters for their games, this time using conditional branching to accommodate different key-events.

Agenda	
20 min	Introduction
20 min	Pizza Toppings
25 min	Player Movement
5 min	Closing

Learning Objectives:

- Reason about the relative positioning of objects using mathematics
- Discover Partial Functions, and how to implement them using `Cond`
- Use Booleans with `cond` to change control flow
- Adapt Design Recipe to add `cond`

Product Outcomes:

- Students will write functions that use conditionals and Booleans
- Students will write `update-player`

State Standards See our Standards Document provided as part of the Bootstrap curriculum.

Length: 90 minutes

Materials and Equipment:

- Computers w/ DrRacket or WeScheme
- Student workbook
- "Luigi's Pizza" [LuigisPizza.rkt from source-files.zip | WeScheme] preloaded on students' machines, in front
- Pens/pencils for the students, fresh whiteboard markers for teachers
- Class posters (List of rules, basic skills, course calendar)
- Language Table (see below)

Preparation:

- Write agenda on board
- All student computers should have their game templates pre-loaded, with their image files linked ins
- Seating arrangements: ideally clusters of desks/tables

Types	Functions
Number	`+ - * / sq sqrt expt`
String	`string-append string-length`
Image	`rectangle circle triangle ellipse radial-star scale rotate put-image`
Boolean	`= > < string=? and or`

Introduction (Time 20 min)

- Have students complete the Warmup activity for Luigi's Pizza, and review their answers.
- So what's special about this code, that makes it different from every other function you've defined?
- Up to now, every function you've defined has done the *same thing* to all of it's inputs. `green-triangle` always made green triangles, no matter what the size was. `safe-left?` always compared the input coordinate to -50, `update-danger` always added or subtracted the same amount, and so on.
- `cost` is special, because different toppings can result in totally different values being returned. That's because `cost` makes use of a special language feature called *conditionals*, which is abbreviated in the code as `cond`.
- Each conditional has at least one *clause*. In Luigi's function, there is a clause for cheese, another for pepperoni, and so on. Every clause has a Boolean question and a result. If the question evaluates to `true`, the expression gets evaluated and returned. If the question is `false`, the computer will skip to the next clause.
- What is the question in the first clause? The second?
- Finally, a conditional can also have an `else` clause, which gets evaluated if all the questions are `false`. In this function, what gets returned if all the questions are false?
- Functions that use conditions are called *piecewise functions*, because each condition defines a separate piece of the function.
- Why do people want piecewise functions? Well, think about the player in your game...you'd like the player to move one way if you hit the "up" key, and another way if you hit the "down" key. Those are two different expressions! Without `cond`, you could only write a function that always moves the player up, or always moves it down - but not both.

Pizza Toppings (Time 20 min)

Intro to Cond Part 1

Intro to Cond Part 2

- Let's take a look at how you might use the Design Recipe to define a piecewise function using Conditionals. Once you know how to use it, you'll be able to write other piecewise functions in your game.
- Turn to the Design Recipe on Page 23 and grab a Design Recipe Worksheet.
- Suppose we've been hired by Luigi's Pizza to write a function that tells us the cost of different pizza pies. Let's use the design recipe to write this function.
- *Have a student read the problem statement.*
- *I need a volunteer to be our function. Pick someone, and copy the contract as they answer. What is your name? cost Your Domain? String Your Range? Number.* `; cost: String -> Number`
- *Can someone from the class tell me how we should call this function? For example, "cost 'cheese'!" What will cost produce? Let's try this with other toppings...*
- Now it's time to write down some examples.
  Can anyone raise their hands and tell me what I'd write?
  `(EXAMPLE (cost "cheese") 9.00)`
- What are some other examples for `cost`? What changes between them? (Answer: The topping and the price returned! Make sure you label those.)
- Do you notice something odd here? This is the first time that we've ever circled something in the second of the examples, which wasn't also circled in the first part. The price that's being produced changes, but the function never takes in the price!
- That's a hint that something special is going on, but let's see how much father the Design Recipe can take us...
- Now for the Function Header. What do I write here? (Answer: `(define (cost topping))`)
- The Function Body is next. But now we don't know what to write! We know that our examples behave differently from one another – sometimes we want to return 9.00, other times it's 10.50, etc. So what do we do? Well, we could fill in all off those results. Let's do that...
  Make a large, 2-column table under the Function Header.
  `(define (cost topping)`

	10.50
	9.00
	11.25
	10.25

)

- But how do we know when we want to produce 9.00? 10.50? (Answer: When the toppings are cheese and pepperoni)
- What we want is a way to go down each line, checking to see if the topping is the right one. If it is, we go on to finish the line. If not, we go on to the next one.
  - What's Domain of our function? (according to the contract) (Answer: String)
  - What's the type of "pepperoni"? (Answer: String)
  - What function compares two strings, and gives back a Boolean? (Answer: `string=?`)
  - What's the Racket code that compares the input topping to the string "pepperoni"? (Answer: `(string=? topping "pepperoni")`)
  - Now we can write that on our first line, as our first topping check. Can you do the rest?
- *Have students fill out the rest of the table*

(string=? topping "pepperoni")	*10.50*
(string=? topping "cheese")	*9.00*
(string=? topping "chicken")	*11.25*
(string=? topping "broccoli")	*10.25*

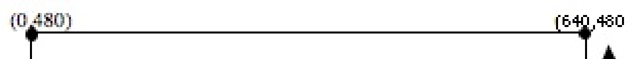
- Each of these rows is called a condition. A condition has a test and a result. The computer goes down the code, one condition at a time, and will evaluate the first result for which the condition is true.
- Racket has a special function that lets us tell the computer to do this: cond. To use cond,you put square brackets around each of the branches, and write "cond" at the top:

```
(define (cost topping)
  (cond
    [(string=? topping "pepperoni") 10.50]
    [(string=? topping "cheese")      9.00]
    [(string=? topping "chicken")    11.25]
    [(string=? topping "broccoli")   10.25]))
```

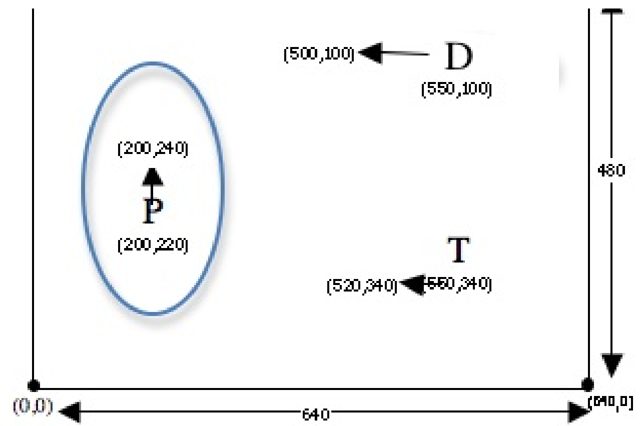
- *Remind students that computers are very specific and can't make up new answers; we need to tell it what to do in case the user inputs an item that is not in our list. Let's add else. If it's not on the menu, we might still make that pizza for you, but it'll cost you!* `[else 10000000]`
- *Have students try it on the computers, adding new items on their own.*
- *If you have additional time, and would like to try another Cond challenge, check out the* [supplemental activity.]

Player Movement (Time 25 min)

- Great! Now that we know `cond` we can write `update-player`.
- *Draw a screen on the board, and label the coordinates for a player, target and danger. Circle all the data associated with the Player.*
- What is the player's starting x-coordinate? (Answer: The player's starting x-coordinate is 200)



- It's starting y-coordinate? (Answer: The player's starting y-coordinate is 220)
- What about after it moves? What's the new x and y? What has changed? And by how much? What happens when we press the down key? What should the new coordinates be then? (Answer: After it moves, its x-coordinate is 200 while its y-coordinate is 240. The x-coordinate has not changed, but its y-coordinate has increased by 20. If you press the down key, the player moves down by 20, so its new coordinate would be (200, 220).)
- *Get students to tell you what update player should do...*
- We want a function that will move up the screen 20 pixels when the user presses the up arrow and down 20 pixels when the user presses the down arrow.



- We've set up the computer to call `update-player`, passing in the player's y-coordinate and the name of the key pressed. The keypress will either be the string `"down"` or the string `"up"` (for now). What kind of data is the y-coordinate? What kind of data is the keypress? (Answer: The y-coordinate is a number and the keypress is a string)
- *Make a table showing possibilities and results, walking students through it.*
- With our pizza example, we had to deal with toppings that weren't on the menu. Now we need to deal with keys that aren't "up" or "down". How do we do that? (Answer: Do nothing! (Or have an else statement that returns the same position))
- On Page 24, you'll find the word problem for `update-player`. Grab a Design Recipe Worksheet, fill it out, and then write this function with your team.
- If you don't like using the arrow keys to make the player move up and down, you can just as easily change them to work with "w" and "x".
- You can also add more advanced movement, by using what you learned about boolean functions. Here's are some ideas..
  - Warping: instead of having the player's y-coordinate change by adding or subtracting, replace it with a Number to have the player suddenly appear at that location. (For example, hitting the "c" key causes the player to warp back to the center of the screen, at y=240.)
  - Wrapping: Add a condition (before any of the keys) that check to see if the player's y-coordinate is above the screen (y > 480). If it is, have the player warp to the bottom (y=0). Add another condition so that the player warps back up to the top of the screen if it moves below the bottom.
  - Challenge: Have the player hide when the "h" key is pressed, only to re-appear when it is pressed again!

Closing (Time 5 min)

- Congratulations - you've got the beginnings of a working game!
- What's still missing? Nothing happens when the player collides with the object or target!
- We're going to fix these over the next few lessons, and also work on the artwork and story for our games, so stay tuned!
- *Who can tell us one thing we learned today? Call on 2-3 volunteers.*
- *Who saw someone else in the class do something great? Call on 2-3 volunteers.*
- *Cleanup, dismissal.*

Bootstrap by Emmanuel Schanzer is licensed under a Creative Commons 3.0 Unported License. Based on a work at www.BootstrapWorld.org. Permissions beyond the scope of this license may be available at schanzer@BootstrapWorld.org.