Directions: Write a function that takes the target's x-coordinate and makes a player leap by returning an x-coordinate that is double the original x-coordinate.

Contract and Purpose Statement

Every contract has three parts…

```
# target-leap :: Number -> Number
```

function name | *domain* | *range*

```
# Takes the x-coordinate and returns a new one, multiplied by 2.
```

what does the function do?

Examples

Write some examples, then circle and label what changes…

```
examples:
  target-leap ( 100 ) is 200
  target-leap ( 40 ) is 200
end
```

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

Definition

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

```
fun leap ( x-coor ):
  x * 5
end
```

function name | *variable(s)* | *what the function does with those variable(s)*

Directions: Write a function, is-offscreen, which returns true is Sam the butterfly's x-coordinate is less than -50 or greater than 690.

Contract and Purpose Statement

Every contract has three parts…

```
# is-offscreen :: Number -> Boolean
```

function name | *domain* | *range*

```
# Given an x-coordinate, returns true if the coordinate is less than -50
```

what does the function do?

Examples

Write some examples, then circle and label what changes…

```
examples:
  is-offscreen ( 60 ) is true
  is-offscreen ( 800 ) is false
end
```

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

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

Definition

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

```
fun is-off-screen ( x-coord ):
  (x-coord < -50) and (x-coord > 690)
end
```

function name | *variable(s)*

what the function does with those variable(s)

Directions: All students are given five (5) pencils at the beginning of the school year. Write a function called calc-pencils that takes in the number of students in the school and calculates the number of pencils needed for that school.

Contract and Purpose Statement

Every contract has three parts…

```
# calc-pencils :: Number -> Number
```

function name | *domain* | *range*

```
# Takes a number of students and gives the number of pencils
```

what does the function do?

Examples

Write some examples, then circle and label what changes…

```
examples:
  calc-pencils ( 100 ) is 100 * 5
  calc-pencils ( 40 ) is 40 * 6
end
```

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

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

Definition

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

```
fun calculate-pencils( p ):
  p * 5
end
```

function name | *variable(s)*

what the function does with those variable(s)

Directions: Write a function that returns the area of a circle given its diameter.

Contract and Purpose Statement

Every contract has three parts…

```
# circle-area :: Number -> Number
```

function name | *domain* | *range*

```
# Given the diameter, multiply pi by radius squared to get the area
```

what does the function do?

Examples

Write some examples, then circle and label what changes…

```
examples:
  circle-area ( 10 ) is num-sqr(10 / 2) * pi
  circle-area ( 50 ) is num-sqr(50 / 2) * pi
end
```

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

Definition

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

```
fun circle-area ( diameter ):
  sqr(diameter) * pi
end
```

function name | *variable(s)*

what the function does with those variable(s)

Directions: It is customary to tip 20% on a bill at a restaurant. Write a function that takes the total cost of the food and returns the new total including tip.

Contract and Purpose Statement

Every contract has three parts…

```
# check-total :: Number -> Number
```
function name / *domain* / *range*

```
# Returns the total of a check with 20% of the cost added
```
what does the function do?

Examples

Write some examples, then circle and label what changes…

```
examples:
  total ( 20 ) is 20)
  total ( 56.67 ) is 56.67)
end
```
function name / *input(s)* / *what the function produces*

Definition

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

```
fun check-total ( food-total ):
  (0.2 + food-total) * food-total
end
```
function name / *variable(s)* / *what the function does with those variable(s)*

Directions: You have 100 square feet of carpet to put down in your room. Write a function that takes in the length and width of your room and returns true if you have enough carpet and false if you don't.

Contract and Purpose Statement

Every contract has three parts…

```
# have-enough-carpet:: Number, Number -> Number
```

function name | *domain* | *range*

```
# Given length and width of a room, is the area <= 100 sq feet?
```

what does the function do?

Examples

Write some examples, then circle and label what changes…

```
examples:
  have-enough-carpet ( (10 15) ) is (10 * 15) < 100
  have-enough-carpet ( (9 10) ) is (9 * 10) < 100
end
```

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

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

Definition

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

```
fun have-enough-carpet(length, width):
  (length * width) < 100
end
```

function name | *variable(s)*

what the function does with those variable(s)

Directions: You go to the store with $1.50 in your pocket. Write a function that takes in the price of an item and returns true if you have enough money to buy the item and false if you do not.

Contract and Purpose Statement

Every contract has three parts…

```
# have-enough-cash:: String -> Boolean
```

function name | *domain* | *range*

```
# Check to see if the item costs less than 1.50
```

what does the function do?

Examples

Write some examples, then circle and label what changes…

```
examples:
  have-enough-cash ( 2.5 ) is 1.50 >= 2.50
  have-enough-cash ( 9.0 ) is gum < 150
end
```

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

Definition

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

```
fun have-enough-cash( item ):
  item <= 1.5
end
```

function name | *variable(s)* | *what the function does with those variable(s)*

Directions: Write a function that takes in two strings and returns true if their lengths are equal and false otherwise.

Contract and Purpose Statement

Every contract has three parts…

```
# equal-length :: String, String -> Boolean
```

function name | *domain* | *range*

```
# Given two strings, check if they are the same length
```

what does the function do?

Examples

Write some examples, then circle and label what changes…

```
examples:
  equal-length ( "yes", "no" ) is string-length("yes") == string-length("no")
  equal-length ( "dog", "cat" ) is string-length("dog") == string-length("cat")
end
```

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

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

Definition

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

```
fun equal-length (string1, string2):
  =
end
```

function name | *variable(s)*

what the function does with those variable(s)

Directions: You are putting together a list of flowers in your garden based on their color. You have red roses, purple tulips, and yellow daisies. Write a function that takes in the color of a flower and returns the name of the flower.

Contract and Purpose Statement

Every contract has three parts…

```
# flower-name :: String -> String
```

function name | *domain* | *range*

```
# Takes the name of the flower and returns its color
```

what does the function do?

Examples

Write some examples, then circle and label what changes…

```
examples:
  flower-name ( "red" ) is rose
  flower-name ( "tulip" ) is purple
  flower-name ( "yellow" ) is daisy
end
```

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

Definition

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

```
fun flower-name ( color ):
  ask:
  | color == "red" then: "rose"
  | color == "purple" then: "tulip"
  | color == "yellow" then: "daisy"
  | otherwise: "That flower isn't in the garden!"
  end
end
```

function name | *variable(s)*

Directions: Names that are longer than 20 characters are considered long names. Write a function that takes in a person's name and returns true if it is a long name and false if it is not.

Contract and Purpose Statement

Every contract has three parts…

```
# is-long-name :: String -> Boolean
```

function name | *domain* | *range*

```
# Check if a name is longer than 20 characters
```

what does the function do?

Examples

Write some examples, then circle and label what changes…

```
examples:
  is-long-name ("\"John Joseph Jingleheimer Schmidt\"")
  is string-equal("John \"Jos Jaime Juarez\"") > 10
  is-long-name ("Juarez\"") is string-equal("Jos Jaime Juarez") > 10
end
```

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

Definition

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

```
fun is-long (name):
  name < 20
end
```

function name | *variable(s)* | *what the function does with those variable(s)*

Directions: Write a function that takes an image and a string, representing what to scale the image by. The function should return a smaller image if the string is 'smaller' and a bigger image if the string is 'bigger'.

Contract and Purpose Statement

Every contract has three parts…

```
# scale-image :: Image, String -> image
```

function name | *domain* | *range*

```
# Make the image bigger or smaller, depending on the given string
```

what does the function do?

Examples

Write some examples, then circle and label what changes…

```
examples:
  scale-image ("circle(5, \"solid\", \"red\")", "\"bigger\"")
  is circle(10, "solid", "red")
  scaleimage ("triangle(20, \"solid\", \"blue\")", "\"smaller\"")
  is triangle(10, "solid", "blue")
end
```

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

Definition

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

```
fun scale-image (original-image, scale-factor):
  ask:
  | scale-factor == "bigger" then: scale(2, original-image)
  | scale-factor == "smaller" then: scale(0.5, original-image)
  | otherwise: original-image
  end
end
```

function name | *variable(s)*

Directions: Some states have different tax rates. New York is 8%, Pennsylvania is 3%, and Delaware is 0%. All other states are 5%. Write a function that takes in the price of an item and returns how much the tax will be on the item.

Contract and Purpose Statement

Every contract has three parts…

```
# state-tax :: String -> Number
```

function name | *domain* | *range*

```
# Given the state and an item's price, return the tax on that item
```

what does the function do?

Examples

Write some examples, then circle and label what changes…

```
examples:
  tax ( "Delaware" ) is 0.0 + price
  tax ( "Georgia" ) is 0.05 + price
end
```

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

Definition

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

```
fun state-tax ( state price ):
  ask:
  | state == "Pennsylvania" then: 0.03 * price
  | state == "New York" then: 0.08 * price
  | state == "Delaware" then: 0.0 * price
  | otherwise: 0.05 * price
  end
end
```

function name | *variable(s)*

Directions: You will be late to class if you have to walk more than 25 pixels to get there. Write a function that takes in your x-coordinate and y-coordinate and the x-coordinate and y-coordinate of the classroom and returns true if you will be late to class and false if you will be on time.

Contract and Purpose Statement

Every contract has three parts…

```
# late-to-class :: Number, Number, Number, Number -> Boolean
```

function name | *domain* | *range*

```
# Takes the coorindates of my location and a classroom and returns true if the distance is more than 25
  and false if it is less than 25.
```

what does the function do?

Examples

Write some examples, then circle and label what changes…

```
examples:
  late-to-class ( 40, 55 ) is 25 > distance(40,55)
  late-to-class ( 40, 55 ) is 25 < distance(40,55)
end
```

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

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

Definition

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

```
fun late-to-class (student-x, student-y, school-x, school-y):
  (25 < distance(student-x, student-y))
end
```

function name | *variable(s)*

what the function does with those variable(s)