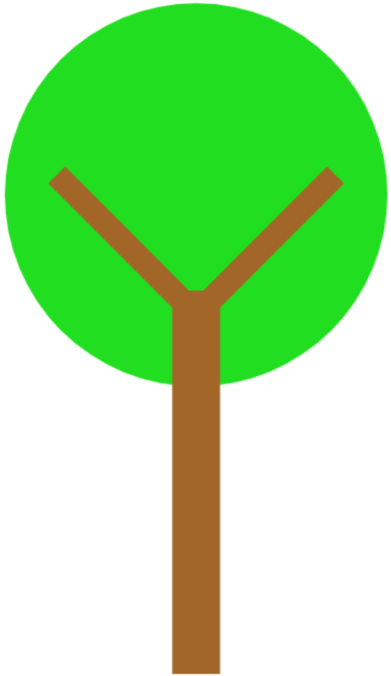
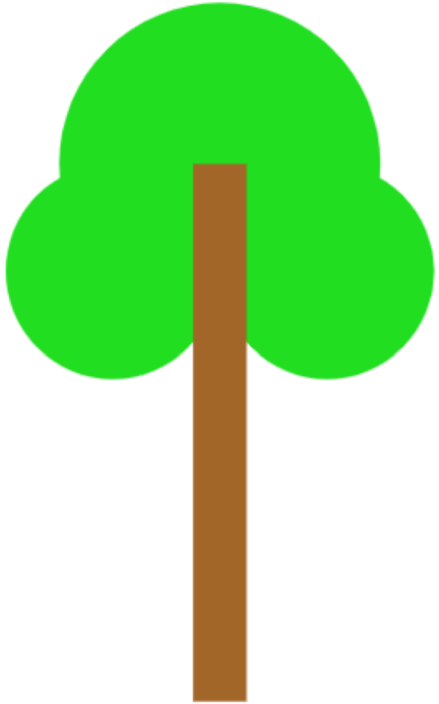
Automated Grading of CodeWorld Exercises

In Practice

Motivation





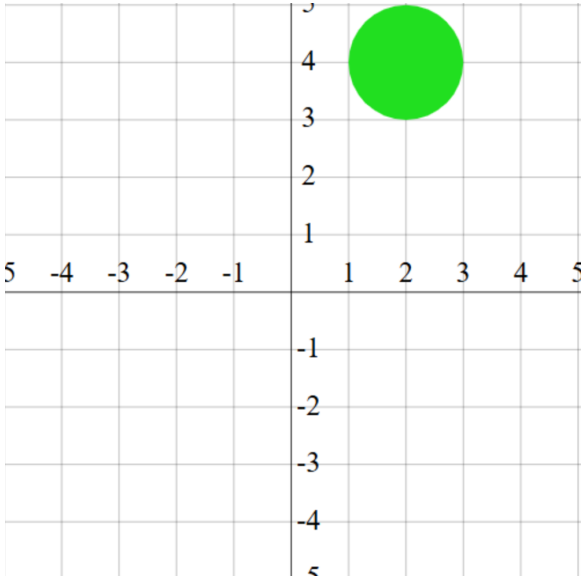
Motivation

```
1  import CodeWorld
2
3
4
5
6  main = drawingOf pic
7
8
9
10 pic = tree
11
12
13
14 tree :: Picture
15 tree = treeTrunk & treeLeaves
16
17 treeTrunk :: Picture
18 treeTrunk = colored brown (solidRectangle 0.1 1)
19
20
21 treeLeaves :: Picture
22 treeLeaves = translated 0 0.5 (colored green (solidCircle 0.3)) &
23   translated 0.2 0.3 (colored green (solidCircle 0.2)) &
24   translated (-0.2) 0.3 (colored green (solidCircle 0.2))
25
26
27
28 {-
29
30 tree :: Picture
31 tree = colored brown (trunk & translated 0 2.5 branches)
32      & translated 0 3 leaves
33
34
35 trunk :: Picture
36
```

Sign In | Guide | Report a Bug | Start Recording | Share | Inspect | Stop | Run

Quick Review

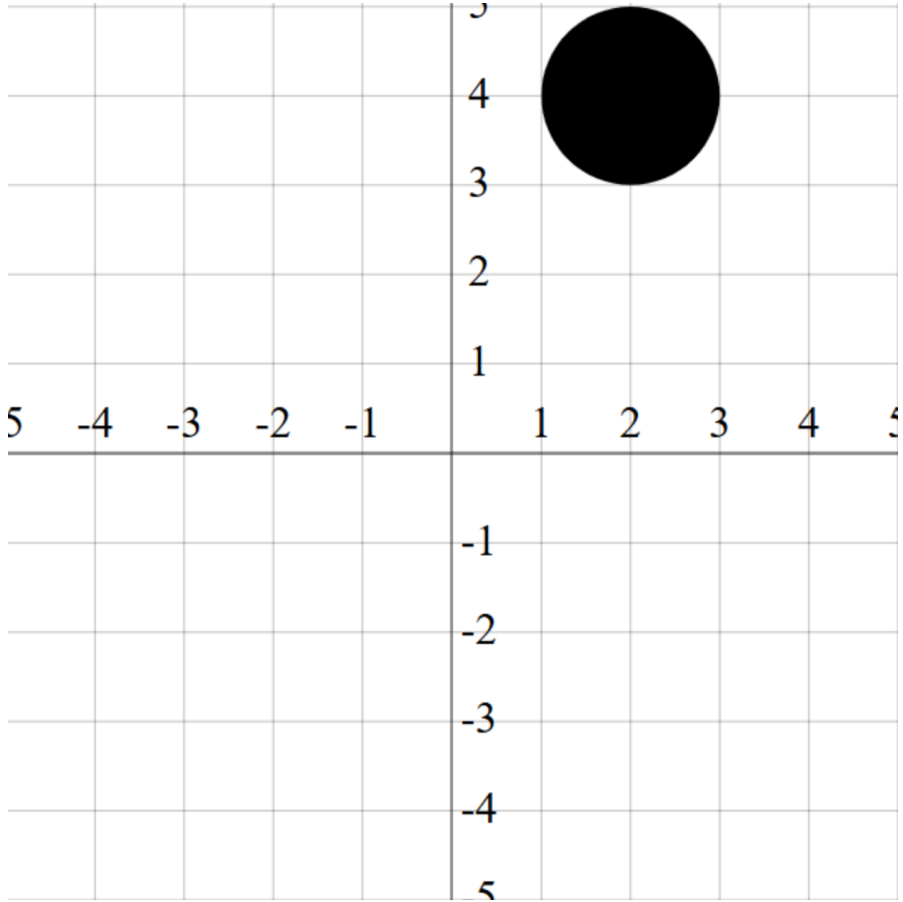
Generated ASTs



translated 2 4 (*colored green* (*solidCircle* 1))

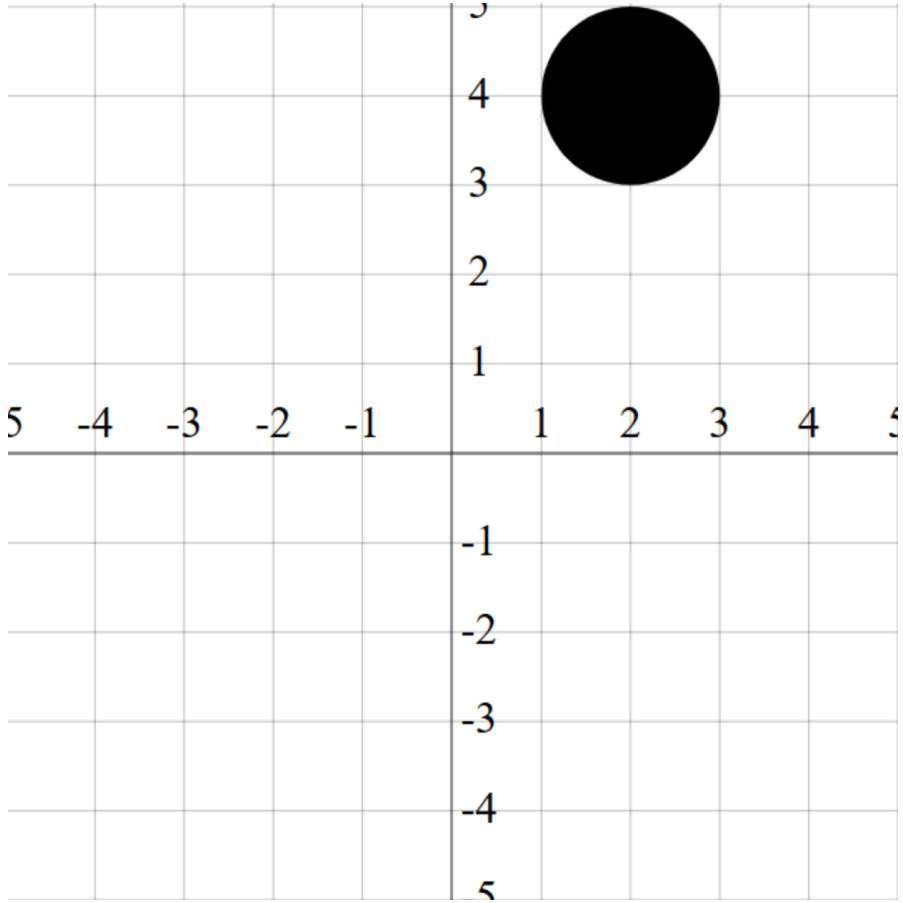
Translate 2 4 (*Color green* (*SolidCircle* 1))

Abstraction



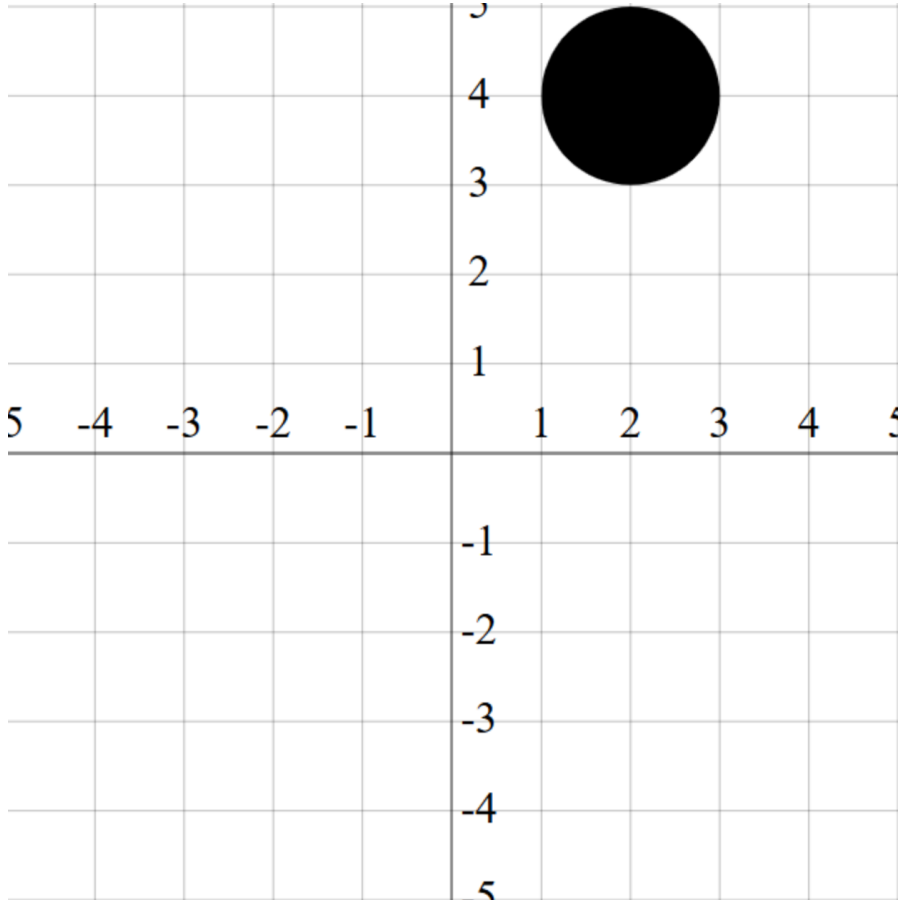
$main = drawingOf\ (translated\ 2\ 4\ (circle\ 2))$

Abstraction



Translate Pos Pos (Circle Size)

Abstraction



“Circle of arbitrary size moved to upper right”

Different ways to draw the same Picture

$solidRectangle\ 1\ 1$

$$solidPolygon\ [(0.5,0.5)\\ ,(0.5,-0.5)\\ ,(-0.5,-0.5)\\ ,(-0.5,0.5)\\ ]$$

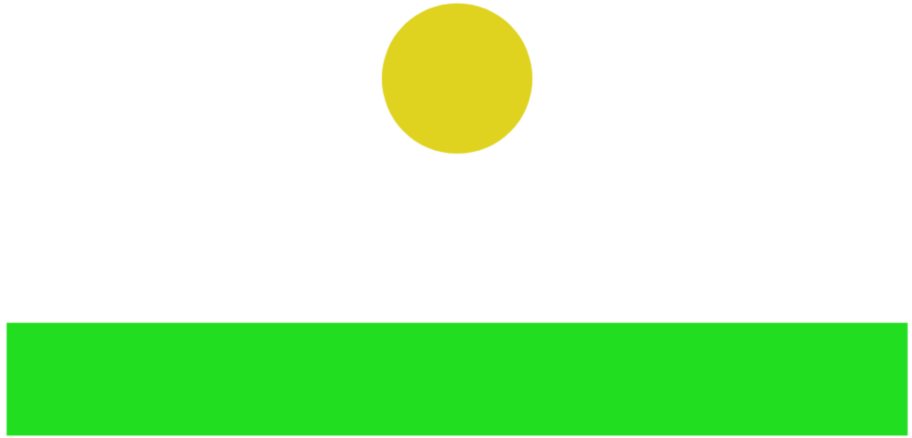
Different ways to draw the same Picture

$solidRectangle\ 1\ 1$

$$solidPolygon\ [(0.5,0.5)\\ ,(0.5,-0.5)\\ ,(-0.5,-0.5)\\ ,(-0.5,0.5)\\ ]$$

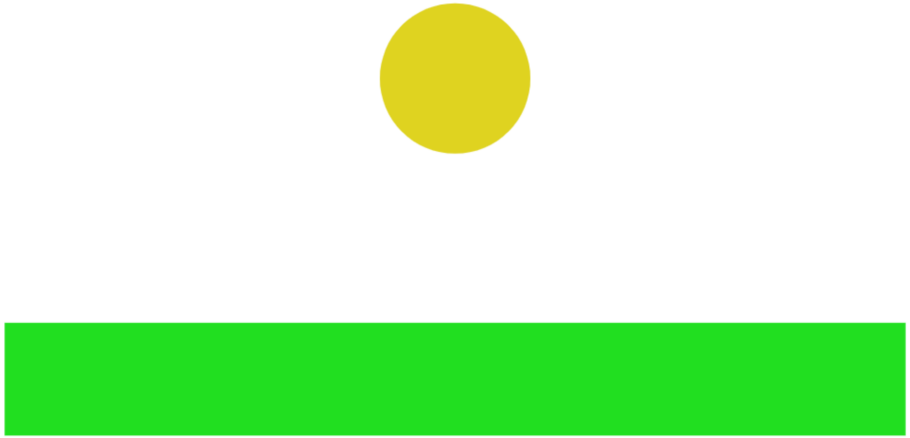
Different ASTs, same picture

Spatial Representation



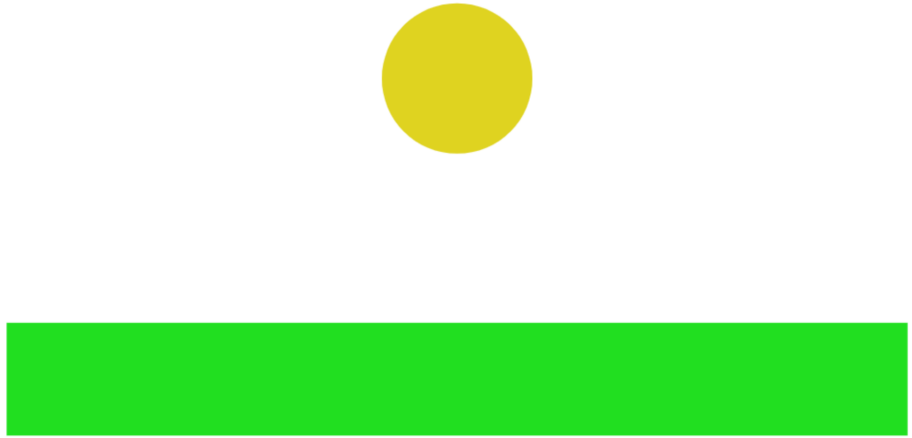
$translated\ 0\ 3\ (colored\ yellow\ (solidCircle\ 1))\ \&\ colored\ green\ (solidRectangle\ 10\ 1)$

Spatial Representation



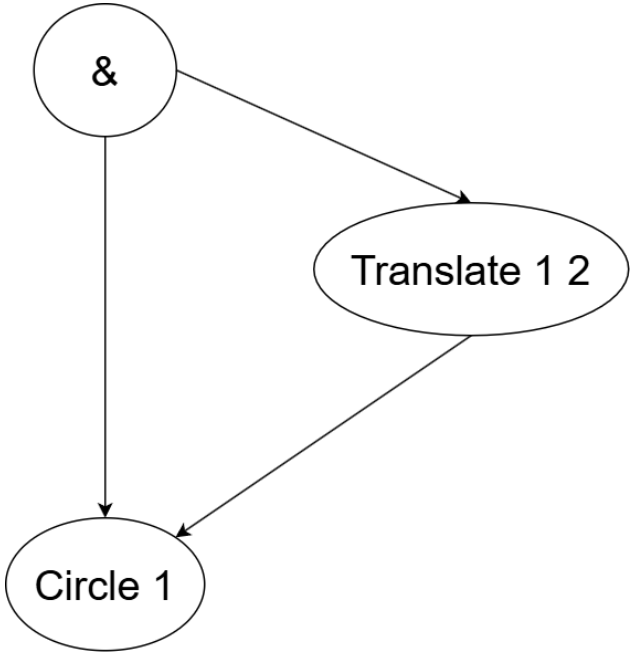
“A yellow circle above a green rectangle”

Spatial Representation



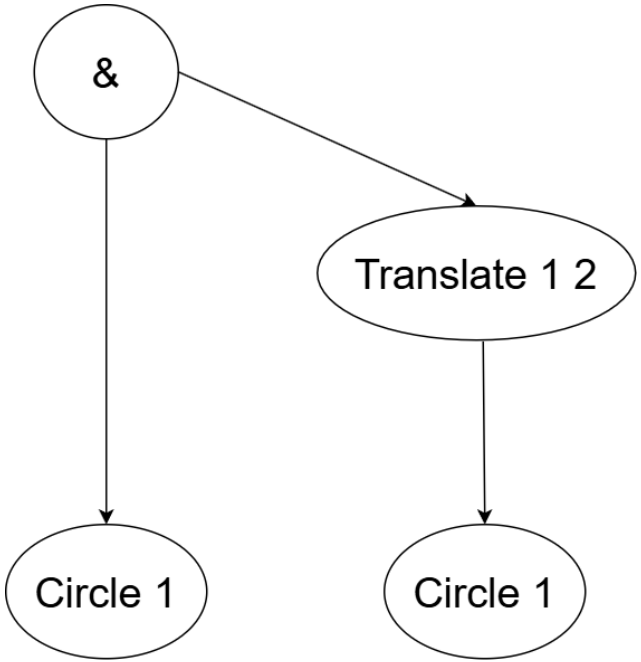
$\text{Color yellow } (\textit{SolidCircle Size}) \text{ `}\boldsymbol{isNorthOf}\text{` Color green } (\textit{SolidRectangle Size Size})$

Reify



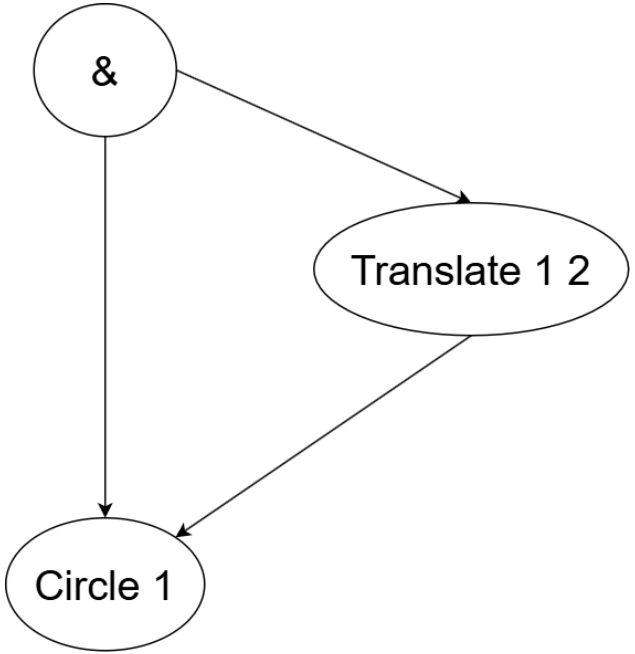

```
let c = circle 1 in c & translated 1 2 c
```

Reify



circle 1 & translated 1 2 (circle 1)

Hashcons



let c = circle 1 in c & translated 1 2 c

circle 1 & translated 1 2 (circle 1)

Summary of Last Talk

Comparing ASTs

- **Abstraction**
- **Normalization**
- **Relative Position**

Unsolved Problems

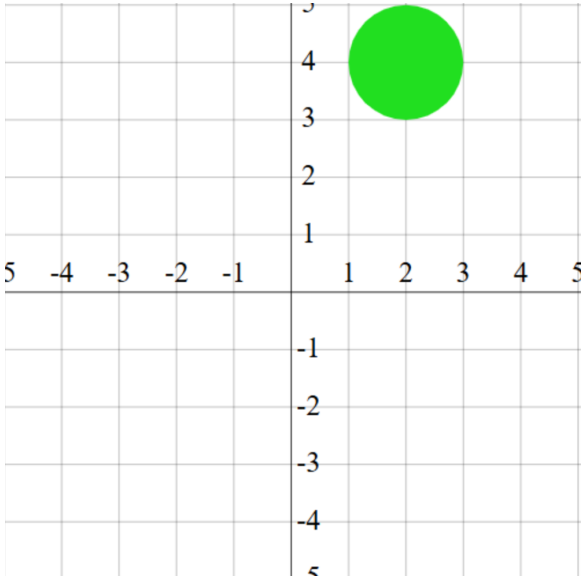
- **Spatial overlap**
- **Solution Spec**
- **Animations**

CSE detection

- **Graph representation**
- **Detect missed sharing opportunities**

Common Tests

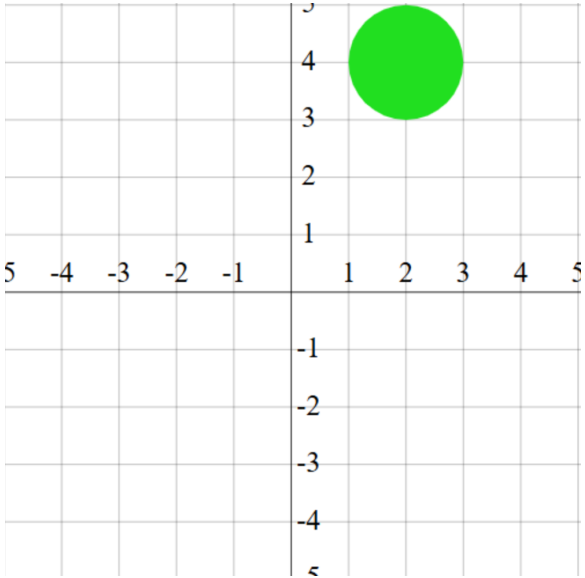
Equal reduced trees



$$translated\ 2\ 4\ (colored\ green\ (solidCircle\ 1))$$

$$colored\ green\ (translated\ 2\ 4\ (solidCircle\ 1))$$

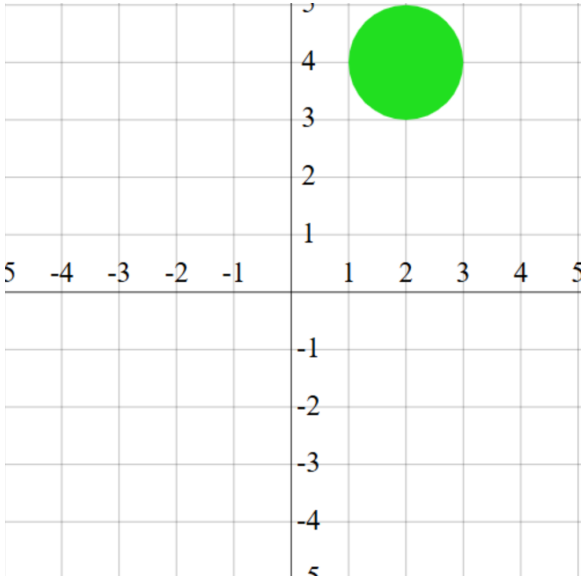
Equal reduced trees



$$translated\ 2\ 4\ (colored\ green\ (solidCircle\ 1))$$

$$translated\ 2\ 4\ (colored\ green\ (rotated\ x\ (solidCircle\ 1)))$$

Equal reduced trees



$$translated\ 2\ 4\ (colored\ green\ (solidCircle\ 1))$$

$$translated\ 2\ 0\ (translated\ 0\ 4\ (colored\ green\ (solidCircle\ 1)))$$

Traversal

- **Most problems can only be solved in a handful of ways**
- **Traverse the un-normalised tree to check some property**
- **Works even if trees are different**

Traversal Example

Traversal Example

NOT WHAT WE WANT!

Equal reduced trees

NOT WHAT WE WANT!

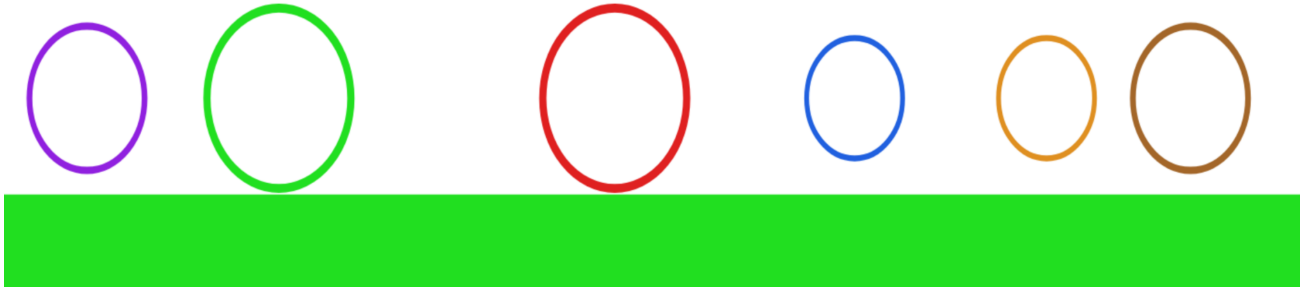
```
moonRotation :: Picture -> Maybe Double
moonRotation p
  | p == Task08.moon = Just 0
  | otherwise = case p of
      Rotate a q  -> (+a) <$> moonRotation q
      Pictures ps -> maximum $ map moonRotation ps
      And p1 p2   -> moonRotation $ Pictures [p1,p2]
      q           -> if hasInnerPicture q
                       then moonRotation (innerPicture q)
                       else Nothing
```

Queries on Normalised Trees

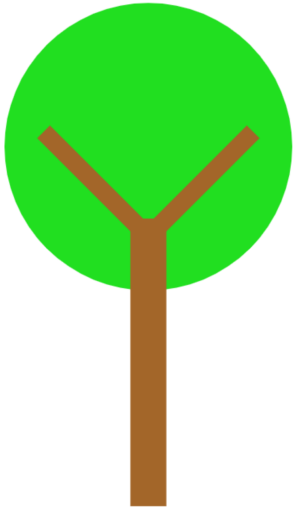
- **Equality does not allow for more sophisticated checks**
- **Retrieve syntax tree’s non-picture arguments**
- **Carry out tests on the results**

Queries on Normalised Syntax Tree

```
let
  usedColors = mapMaybe getColor sceneEggs
  sceneEggs = findAll isEgg $ getComponents Task07.scene
in
  length (nubOrd usedColors) >= 6 ~?
  "There is at least 6 differently colored eggs?"
```

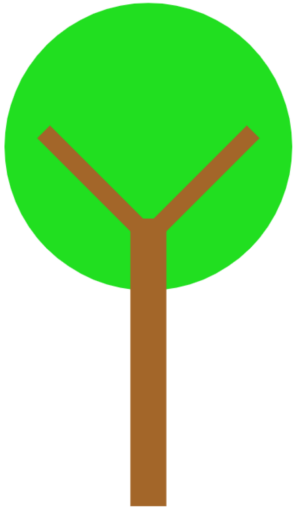


Spatial Queries



```
onScene (hasExactly $ colored green someSolidCircle `northOf` uprightWood) ~?
"The trunk stands upright and there is a tree crown above it?"
```

Spatial Queries



```
onScene (hasExactly $ colored green someSolidCircle `northOf` uprightWood) ~?
"The trunk stands upright and there is a tree crown above it?"
```

```
onSceneMulti
  [ hasBroadly $ rotatedQuarter uprightWood `isAbove` uprightWood
  , hasBroadly $ rotatedUpToFull uprightWood `isAbove` uprightWood
  , hasBroadly $ rotatedQuarter uprightWood `isLeftOf` rotatedUpToFull uprightWood
  ] ~?
"Branches are in the correct position? " ++
"(They should neither cross, nor be detached from the trunk, nor be at level with the trunk)"
```

How to Test Animations?

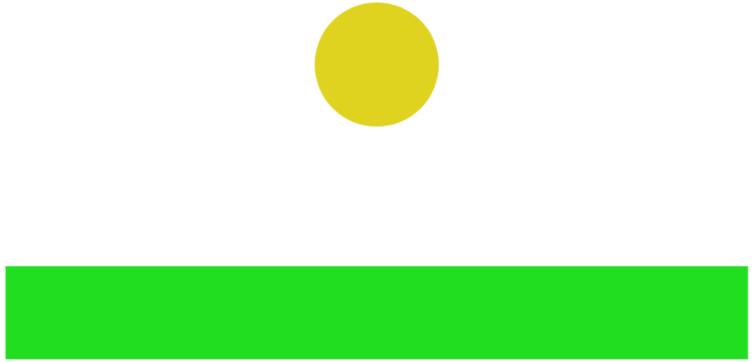
- Sample animation function at set of discrete points
- Do the same as before but with quantifiers
- Problem: need to decide on a “good” sampling frequency

Code Inspection

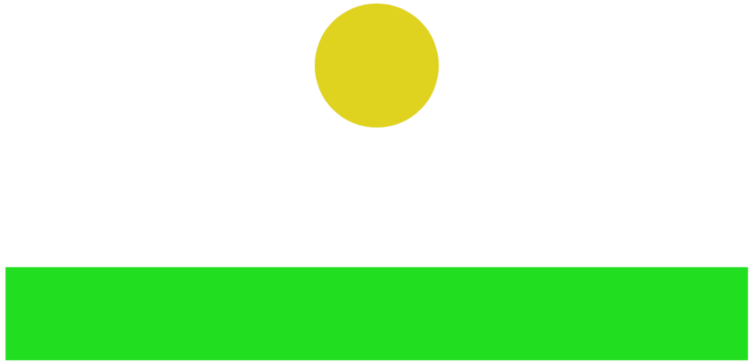
Outline

- Parse source code as syntax tree
- Inspect tree for function use, structure, language features
- Reject immediately on task constraint violations

Required functions



Required functions



```
TestCase $ syntaxCheck $ \m -> assertBool
  ( "The sun and moon are not exhibiting circular movement. " ++
    "Please make sure they are not following a parabola shape or other different motions."
  ) $
  any (\name -> contains (ident name) m) ["sin","cos","rotated"]
```

Structure

```
scene :: Double -> Picture
scene t = grass &
    if t < pi
      then rotated (-t) (translated (-6) 0 sun)
      else blank
  where
    grass = colored green (solidRectangle 20 2)
    sun = colored yellow (solidCircle 1)
```

Structure

```
scene :: Double -> Picture
scene t
  | t < pi = grass &
             rotated (-t) (translated (-6) 0 sun)
  | otherwise = scene t
  where
    grass = colored green (solidRectangle 20 2)
    sun = colored yellow (solidCircle 1)
```

Structure

```haskell
sum :: [Integer] -> Integer
sum =
  divideAndConquer
  undefined
    -- the predicate to test whether an input is simple enough,
    -- should be of type [Integer] -> Bool here
  undefined
    -- the function to compute the output directly for simple cases,
    -- should be of type [Integer] -> Integer here
  undefined
    -- the function to split a non-simple input into two smaller ones,
    -- should be of type [Integer] -> ([Integer], [Integer]) here
  undefined
    -- the function to combine the outputs of two subcomputations,
    -- should be of type Integer -> Integer -> Integer here
```

Structure

```haskell
sum :: [Integer] -> Integer
sum =
  divideAndConquer
  undefined
    -- the predicate to test whether an input is simple enough,
    -- should be of type [Integer] -> Bool here
  sum
    -- the function to compute the output directly for simple cases,
    -- should be of type [Integer] -> Integer here
  undefined
    -- the function to split a non-simple input into two smaller ones,
    -- should be of type [Integer] -> ([Integer], [Integer]) here
  undefined
    -- the function to combine the outputs of two subcomputations,
    -- should be of type Integer -> Integer -> Integer here
```

Disallowed/Enforced Features

- **Task explicitly states the use of some language feature is required**
- **Students just ignore the constraint**
- **Solution: Look for corresponding constructor in program's syntax tree**

Issues

Sampling Phase

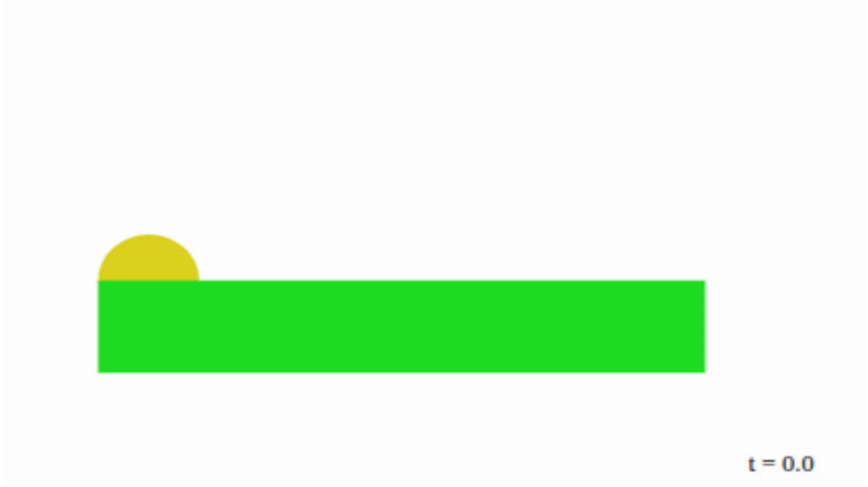
- Sampling could overlap with submission’s period
- We might miss what we are looking for
- Solution: use some irregular pattern for selecting samples

Sampling Frequency Speed

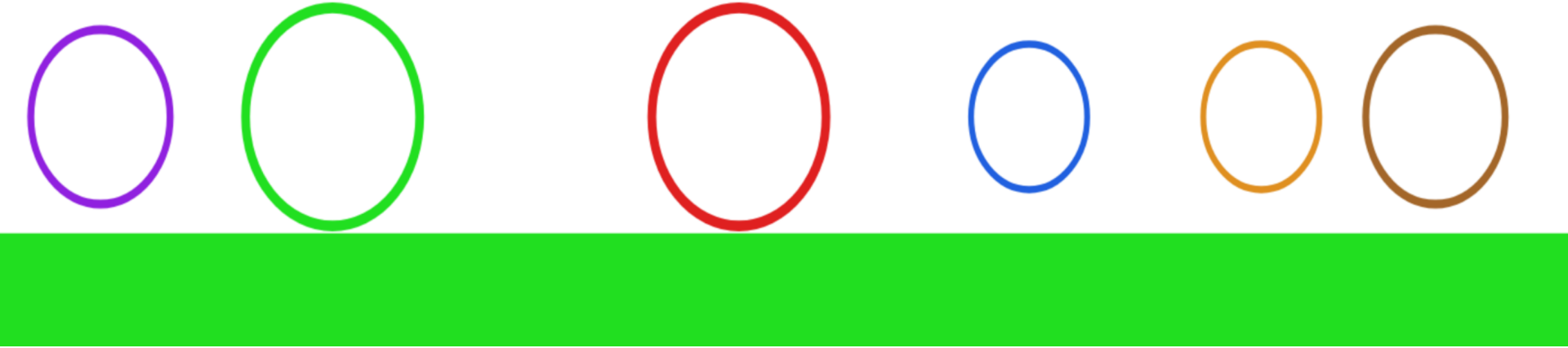
Ad Hoc Additions

- **Correct submission is not accepted for unknown reasons**
- **Add another rule to normalization**
- **… Update library and re-deploy**

False Positives



CSE Output



CSE Output

```
scene :: Picture
scene = translated 0 (-3) (colored green (solidRectangle 20 3)) &
        pictures [ colored f (dilated d (translated x 0 egg))
                 | (x, d, f) <- [ (-9, 1, green)
                                , (5, 1.2, red)
                                , (-5, 0.5, blue)
                                , (-4.5, 0.9, grey)
                                , (-2.5, 0.3, black)
                                , (9, 1, yellow)
                                , (3, 0.2, orange)
                                , (5, 0.2, light red)
                                , (7, 0.1, light blue)
                                , (9, 0.3, light green)
                                ]
                 ]
  where
    egg :: Picture
    egg = scaled 1 1.25 (colored grey (thickCircle 0.2 1))
```

CSE Output

```
let
  aScaledColoredGreyThickCircle = scaled 1.0 1.25 (colored grey (thickCircle 0.2 1.0))
in
  translated 0.0 (-3.0) (colored green (solidRectangle 20.0 3.0)) &
  pictures
    [ colored green (dilated 1.0 (translated (-9.0) 0.0 aScaledColoredGreyThickCircle))
    , colored red (dilated 1.2 (translated 5.0 0.0 aScaledColoredGreyThickCircle))
    , colored blue (dilated 0.5 (translated (-5.0) 0.0 aScaledColoredGreyThickCircle))
    , colored grey (dilated 0.9 (translated (-4.5) 0.0 aScaledColoredGreyThickCircle))
    , colored black (dilated 0.3 (translated (-2.5) 0.0 aScaledColoredGreyThickCircle))
    , colored yellow (dilated 1.0 (translated 9.0 0.0 aScaledColoredGreyThickCircle))
    , colored orange (dilated 0.2 (translated 3.0 0.0 aScaledColoredGreyThickCircle))
    , colored (light red) (dilated 0.2 (translated 5.0 0.0 aScaledColoredGreyThickCircle))
    , colored (light blue) (dilated 0.1 (translated 7.0 0.0 aScaledColoredGreyThickCircle))
    , colored (light green) (dilated 0.3 (translated 9.0 0.0 aScaledColoredGreyThickCircle))
    ]
```

CSE Output

```
It could be rewritten like this:

let
  aScaledColoredGreyThickCircle = scaled 1.0 1.25 (colored grey (thickCircle 0.2 1.0))
  aTranslatedScaledColoredGreyThickCircle = translated 5.0 0.0 aScaledColoredGreyThickCircle
  aTranslatedScaledColoredGreyThickCircle = translated 9.0 0.0 aScaledColoredGreyThickCircle
in
  translated 0.0 (-3.0) (colored green (solidRectangle 20.0 3.0)) &
  pictures
    [ colored green (dilated 1.0 (translated (-9.0) 0.0 aScaledColoredGreyThickCircle))
    , colored red (dilated 1.2 aTranslatedScaledColoredGreyThickCircle)
    , colored blue (dilated 0.5 (translated (-5.0) 0.0 aScaledColoredGreyThickCircle))
    , colored grey (dilated 0.9 (translated (-4.5) 0.0 aScaledColoredGreyThickCircle))
    , colored black (dilated 0.3 (translated (-2.5) 0.0 aScaledColoredGreyThickCircle))
    , colored yellow (dilated 1.0 aTranslatedScaledColoredGreyThickCircle)
    , colored orange (dilated 0.2 (translated 3.0 0.0 aScaledColoredGreyThickCircle))
    , colored (light red) (dilated 0.2 aTranslatedScaledColoredGreyThickCircle)
    , colored (light blue) (dilated 0.1 (translated 7.0 0.0 aScaledColoredGreyThickCircle))
    , colored (light green) (dilated 0.3 aTranslatedScaledColoredGreyThickCircle)
    ]
```

Summary

Mostly worked out

- Test format
- Handling animations

Points to work on

- Fine-tuning the sampling
- Sorting out false positives
- CSE output