EECS 111

A simple **2D graphics** language

One bit of magic

(**require** 2htdp/image)

- None of the graphics procedures are defined by default
- To access the library of graphics procedures, you have to add a **require declaration** at the beginning of your file
  - Pass it the argument - the name of the library
  - In this case, **2htdp/image**
    - 2htdp means the code for the second edition of *How to Design Programs*
    - Image means the image library for 2htdp
- Put it at the **top of your file (definitions pane)**

Procedures that make images

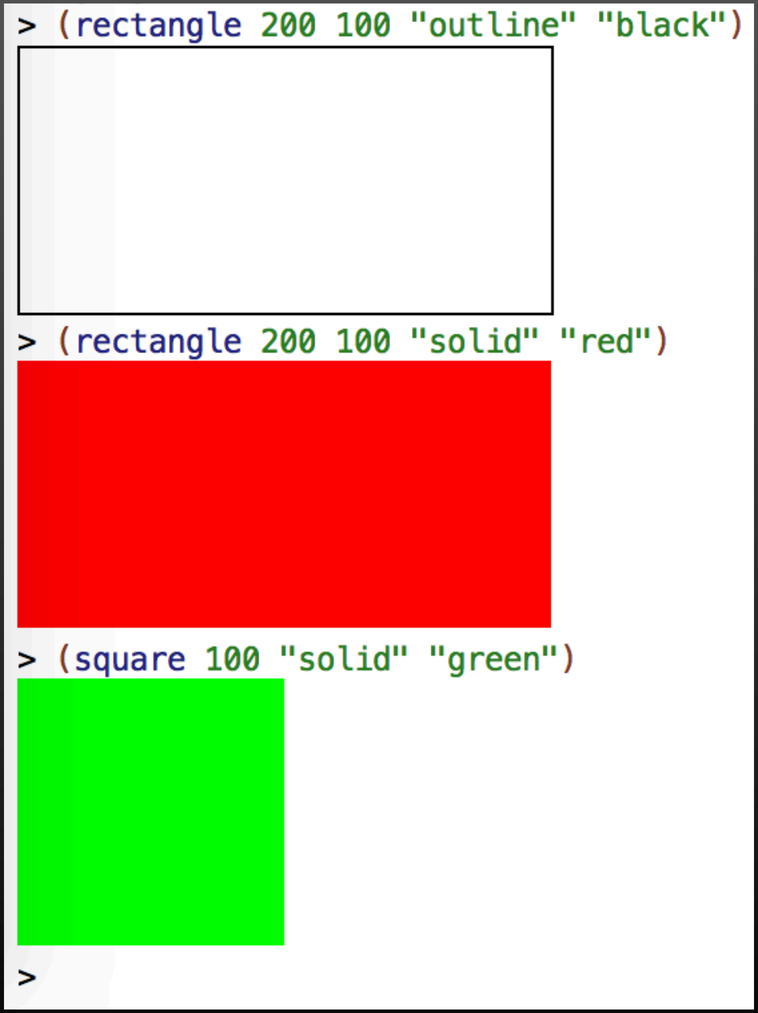
- (rectangle *width height mode color*)
  (square *width mode color*)
  - An image of a rectangle with the specified *width* and *height* (which must be numbers)
  - *Mode* must be either the string **"solid"** or the string **"outline"**
  - *Color* is either the name of a color (e.g. "red", "blue", etc.) or a **color object** (we'll get to these shortly)
- (ellipse *width height mode color*)
  (circle *width mode color*)
  - Same, but gives you a curved object instead of a rectangle

Rectangles

(rectangle *width height*
mode color)

(square *width mode color*)

- Creates a rectangle
  - Width is *width*
  - Height is *height*
  - *Mode* must be either the string **"solid"** or the string **"outline"**
  - *Color* is either the name of a color (e.g. "red", "blue", etc.) or a **color object** (we'll get to these shortly)



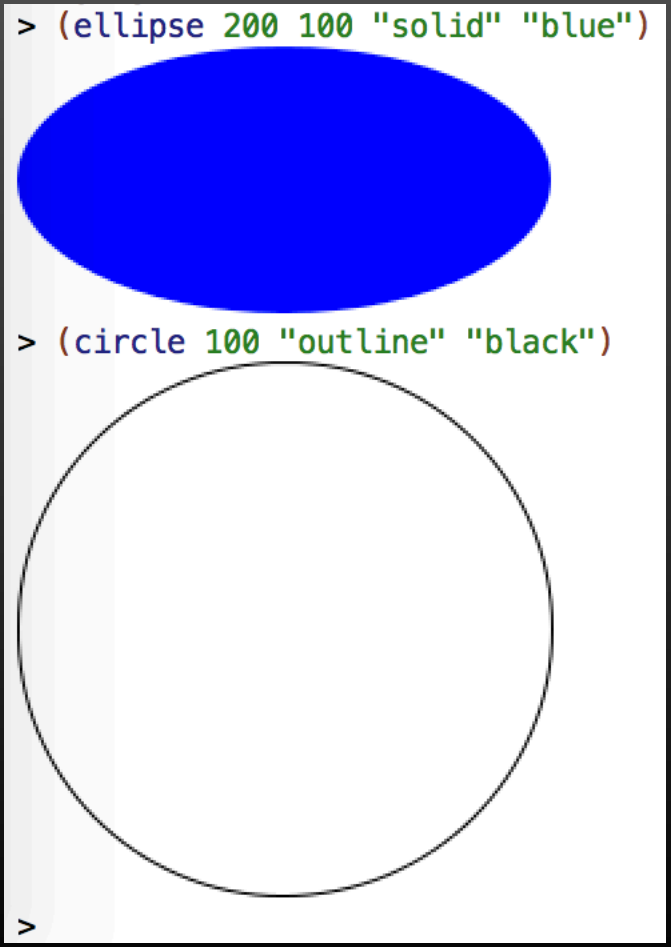
```
> (rectangle 200 100 "outline" "black")
> (rectangle 200 100 "solid" "red")
> (square 100 "solid" "green")
>
```

Circles and ellipses

(ellipse *width height*
mode color)

(circle *width mode color*)

- Same, but curved
  - If *width* and *height* are the same, you get a circle
  - Otherwise, an ellipse

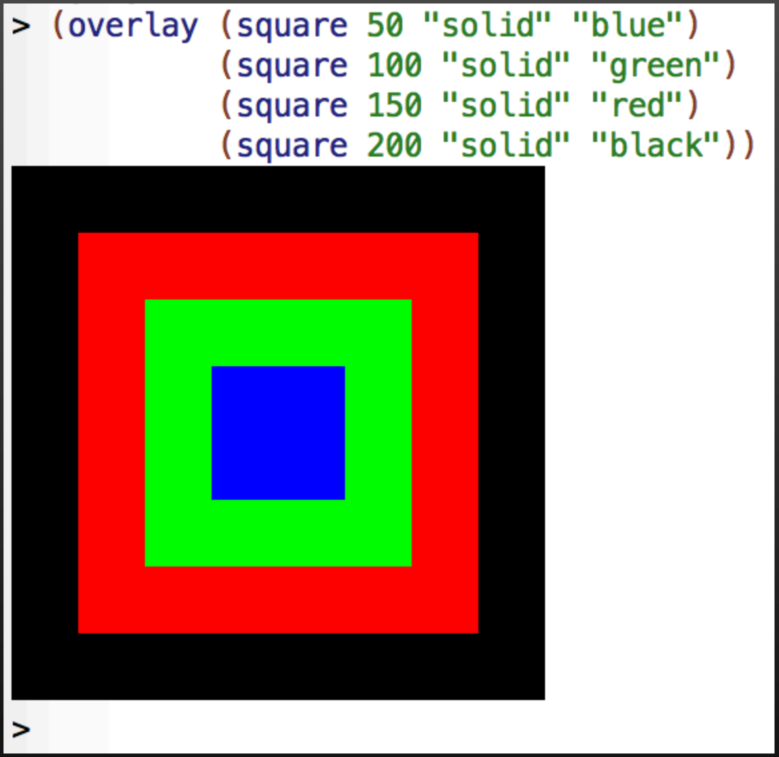


```
> (ellipse 200 100 "solid" "blue")
> (circle 100 "outline" "black")
>
```

Compositing images

(overlay *image image …*)
(underlay *image image …*)

- Make a composite image from multiple image objects
- *images* are **drawn on top of one another**
  - With **earlier ones** being **on top** of later ones (overlay)
  - Or **below** later ones (underlay)
- *image* objects can be shapes or other groups



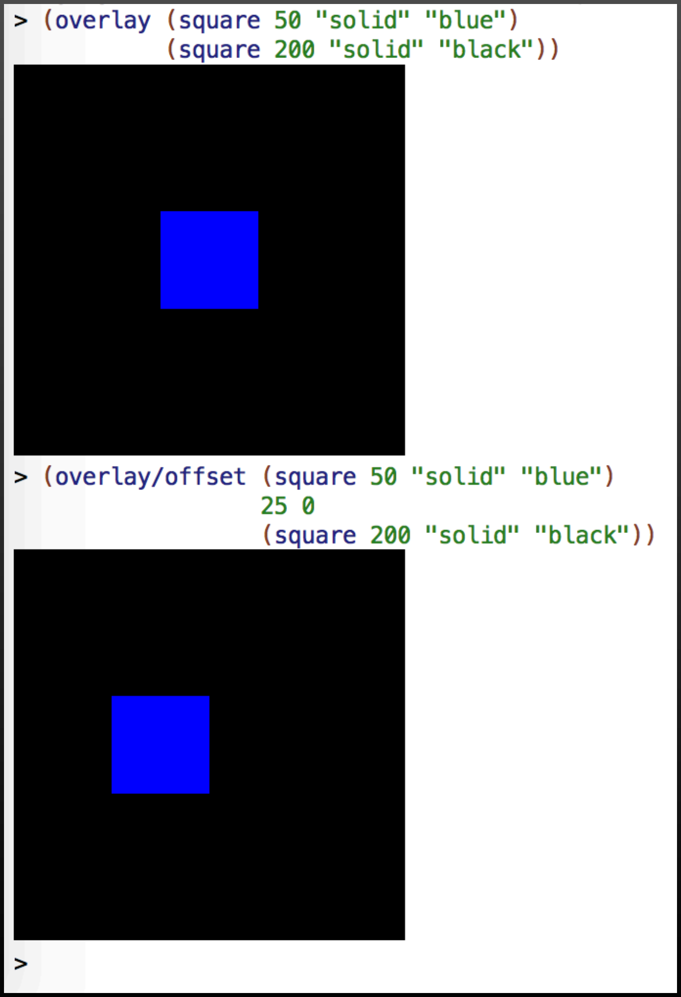
```
> (overlay (square 50 "solid" "blue")
           (square 100 "solid" "green")
           (square 150 "solid" "red")
           (square 200 "solid" "black"))
```

```
>
```

Shifting images relative to one another

(overlay/offset *top-image*
right down
bottom-image)

- Composites the two images
- The *bottom-image* is shifted the specified number of pixels *right* and *down* relative to the top image
  - You can also think of its has moving the top image left and up relative to the bottom image



```
> (overlay (square 50 "solid" "blue")
           (square 200 "solid" "black"))
```

```
> (overlay/offset (square 50 "solid" "blue")
                  25 0
                  (square 200 "solid" "black"))
```

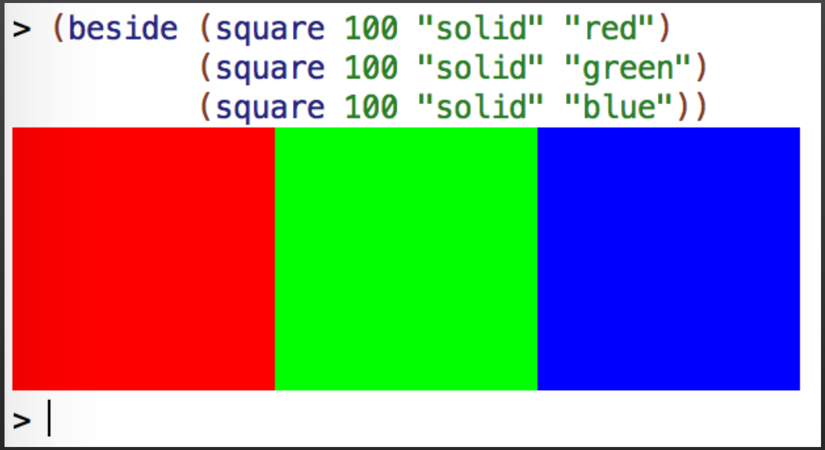
```
>
```

Placing images next to one another

(above *images* …)

(beside *images* …)

- Makes an image by laying out the specified images vertically or horizontally



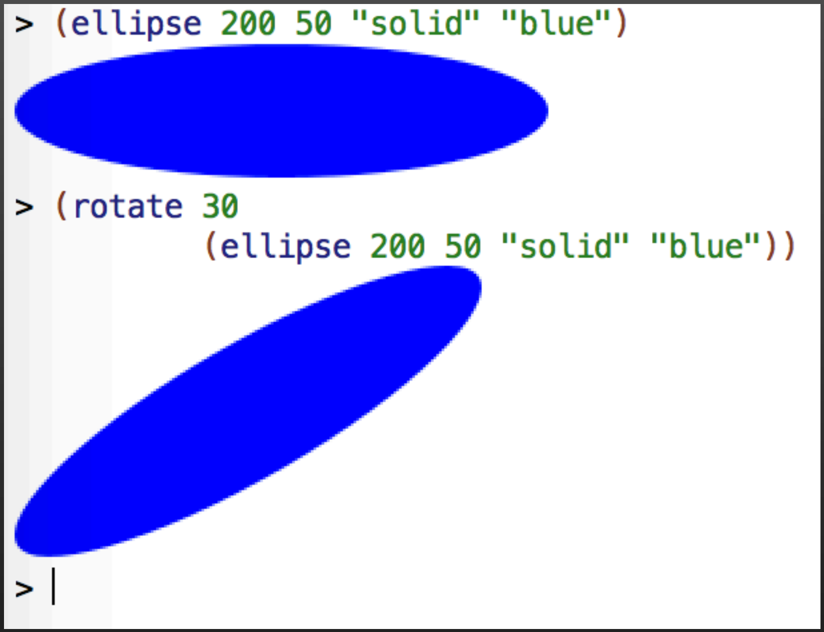
```
> (beside (square 100 "solid" "red")
          (square 100 "solid" "green")
          (square 100 "solid" "blue"))
```

```
> |
```

Rotate

(rotate *angle image*)

- Rotates *image* counter-clockwise *angle* degrees about its center



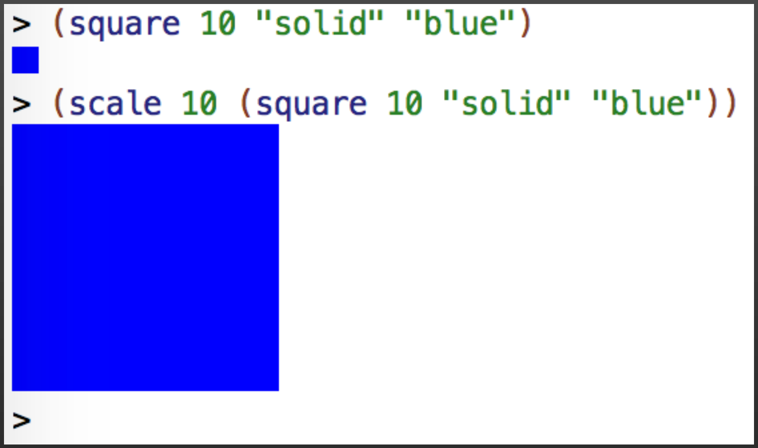
```
> (ellipse 200 50 "solid" "blue")

> (rotate 30
          (ellipse 200 50 "solid" "blue"))

> |
```

Scaling

(scale *scale-factor image*)



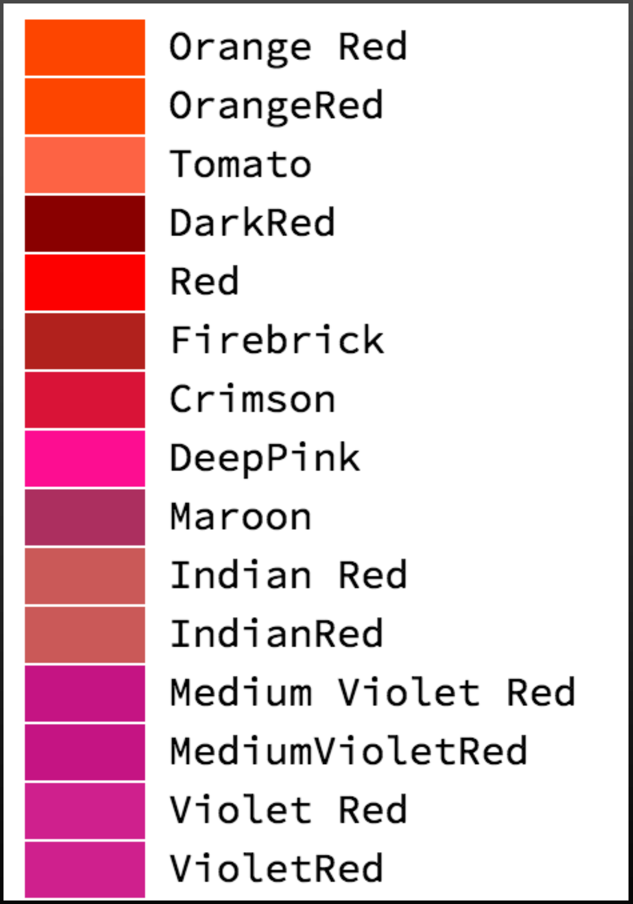
```
> (square 10 "solid" "blue")

> (scale 10 (square 10 "solid" "blue"))

>
```

Colors

- Colors can be specified by name using **strings**
- Racket understands the names for the standard **primaries** and **secondaries**
  - “red”, “green”, “blue”, “cyan”, “magenta”, “yellow”
- As well as the standard Web colors and X11 color names

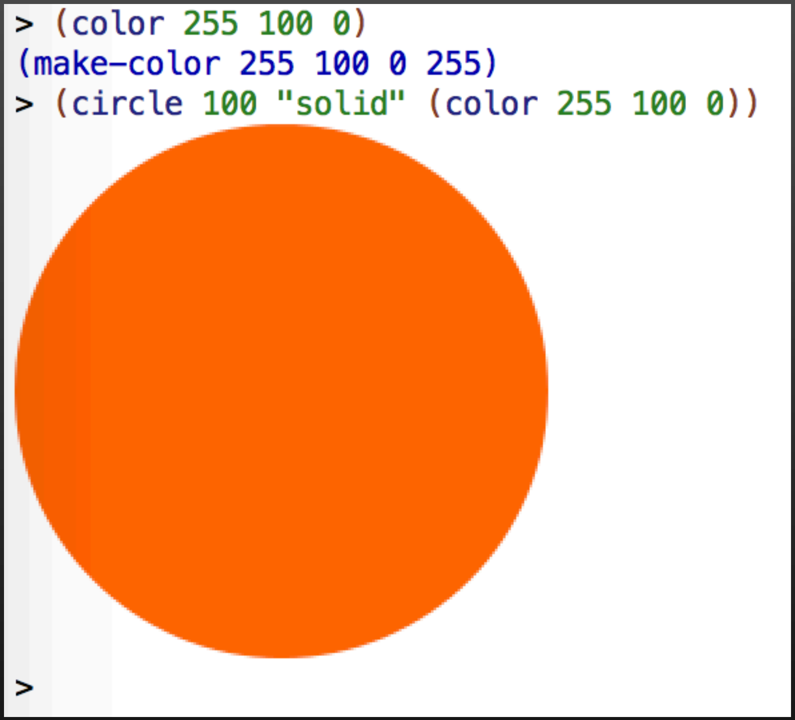


🟧	Orange Red
🟧	OrangeRed
🟧	Tomato
🟥	DarkRed
🟥	Red
🟥	Firebrick
🟥	Crimson
🟪	DeepPink
🟪	Maroon
🟥	Indian Red
🟥	IndianRed
🟪	Medium Violet Red
🟪	MediumVioletRed
🟪	Violet Red
🟪	VioletRed

Color objects

(make-color *r g b*)
(make-color *r g b a*)
(color *r g b*)

- You can also specify a color using a color object
  - Specifies the amount of red, green, and blue light in the color, on a 0-255 scale
- Note: when Racket prints a color object, it prints it textually, not as the color itself



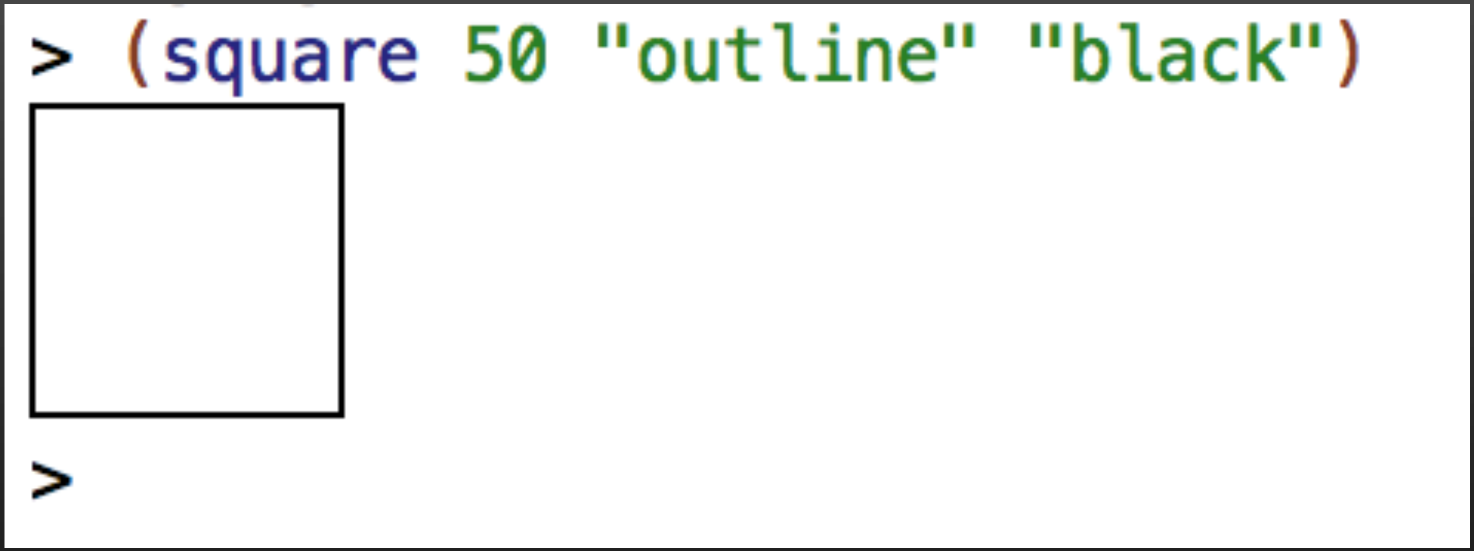
```
> (color 255 100 0)
(make-color 255 100 0 255)
> (circle 100 "solid" (color 255 100 0))
```

```
>
```

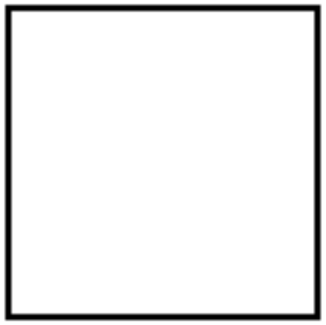
Rules of computation in Racket

- If it's a **constant** (a number or string)
  - It's its own value
  - Return it
- If it's a **variable name** (e.g. a word, hyphenated-phrase, or e.g. symbol)
  - Look up its value in the **dictionary**
  - Return (output) it
- If it has **parens** (i.e. it looks like "(*a b c* …)")
  - Find the values of *a*, *b*, *c*, etc. using these same rules
  - The value of *a* had **better be a procedure**
  - Call it with the values of *b*, *c*, etc. as inputs
  - Return its output

How do we make a **square**?

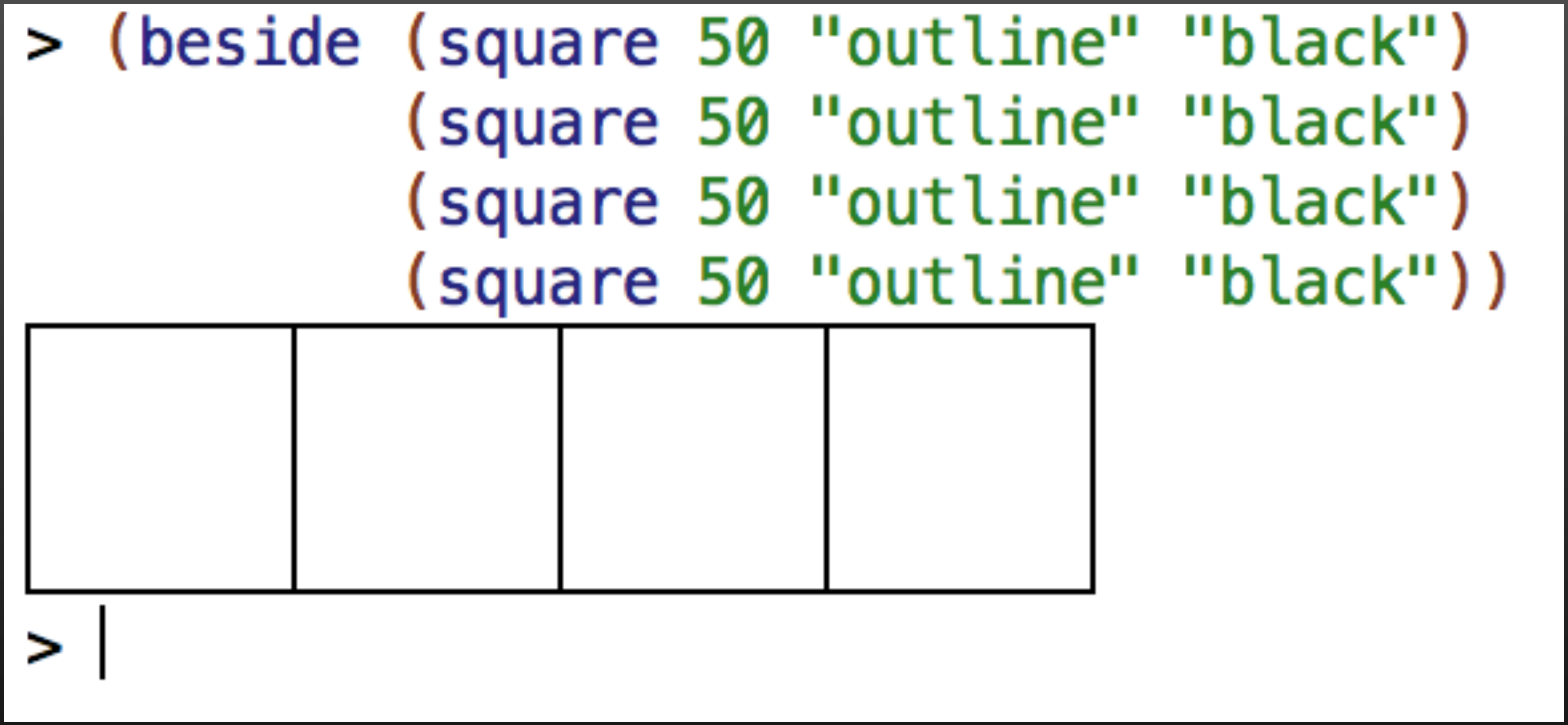


```
> (square 50 "outline" "black")
```

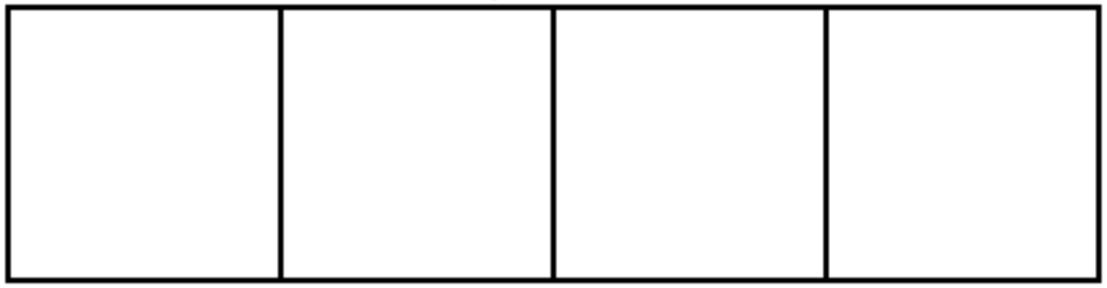


```
>
```

How do we make a **row of squares**?

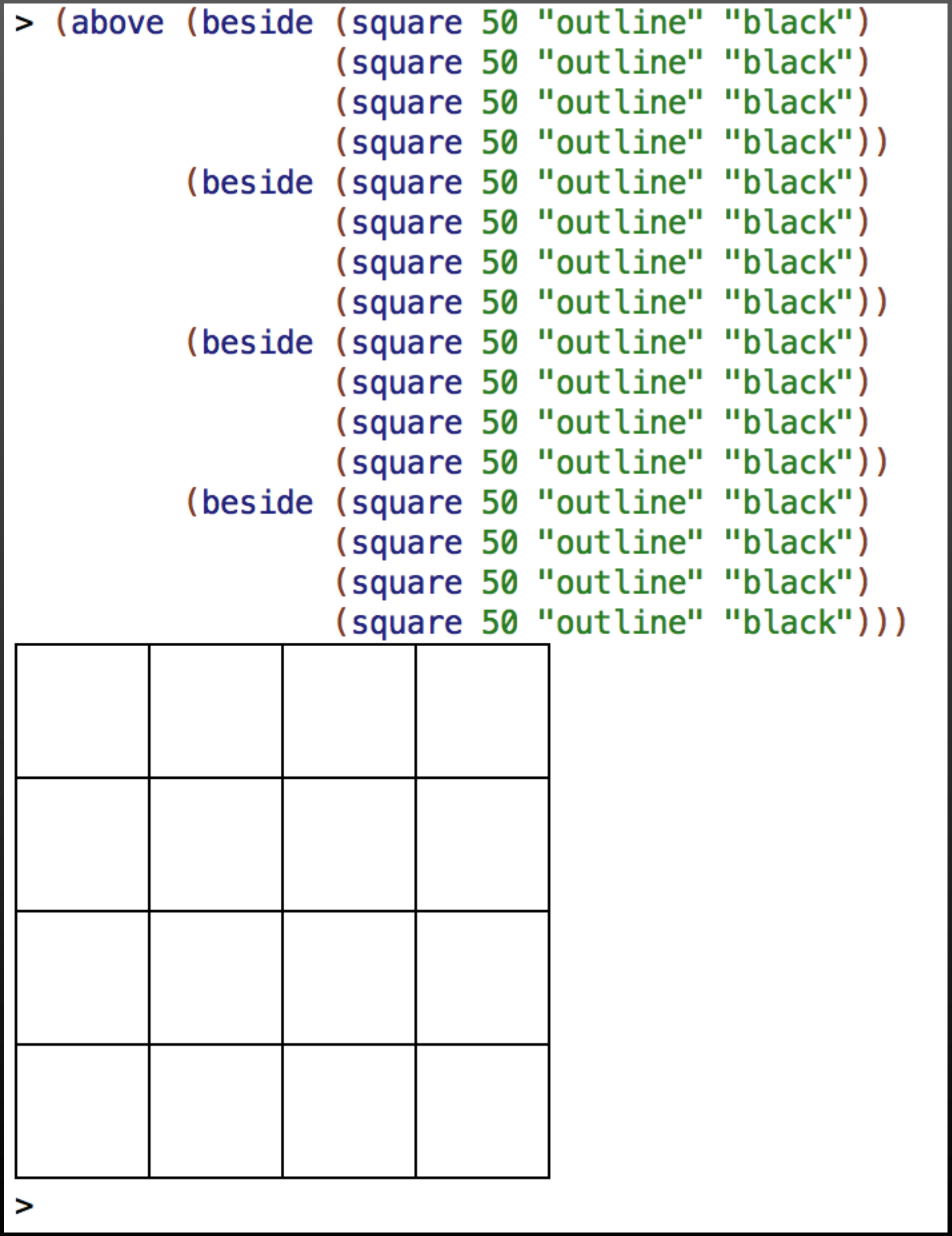


```
> (beside (square 50 "outline" "black")
          (square 50 "outline" "black")
          (square 50 "outline" "black")
          (square 50 "outline" "black"))
```

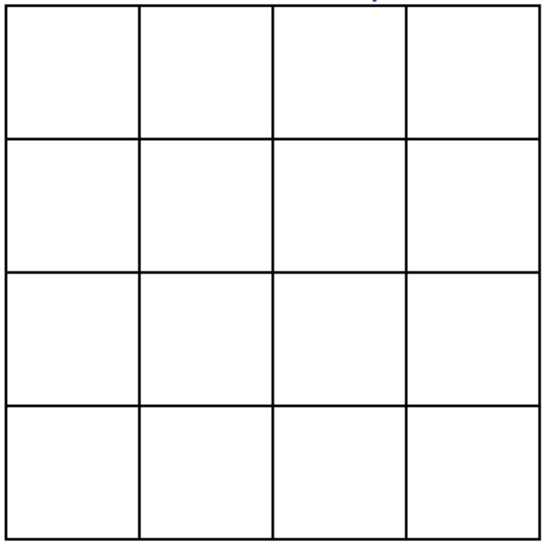


```
> 
```

How do we make a **grid**?



```
> (above (beside (square 50 "outline" "black")
                 (square 50 "outline" "black")
                 (square 50 "outline" "black")
                 (square 50 "outline" "black"))
         (beside (square 50 "outline" "black")
                 (square 50 "outline" "black")
                 (square 50 "outline" "black")
                 (square 50 "outline" "black"))
         (beside (square 50 "outline" "black")
                 (square 50 "outline" "black")
                 (square 50 "outline" "black")
                 (square 50 "outline" "black"))
         (beside (square 50 "outline" "black")
                 (square 50 "outline" "black")
                 (square 50 "outline" "black")
                 (square 50 "outline" "black")))
```



```
>
```

Wow, that’s a lot of typing…

```
(above (beside (square 50 "outline" "black")
               (square 50 "outline" "black")
               (square 50 "outline" "black")
               (square 50 "outline" "black"))
       (beside (square 50 "outline" "black")
               (square 50 "outline" "black")
               (square 50 "outline" "black")
               (square 50 "outline" "black"))
       (beside (square 50 "outline" "black")
               (square 50 "outline" "black")
               (square 50 "outline" "black")
               (square 50 "outline" "black"))
       (beside (square 50 "outline" "black")
               (square 50 "outline" "black")
               (square 50 "outline" "black")
               (square 50 "outline" "black")))
```

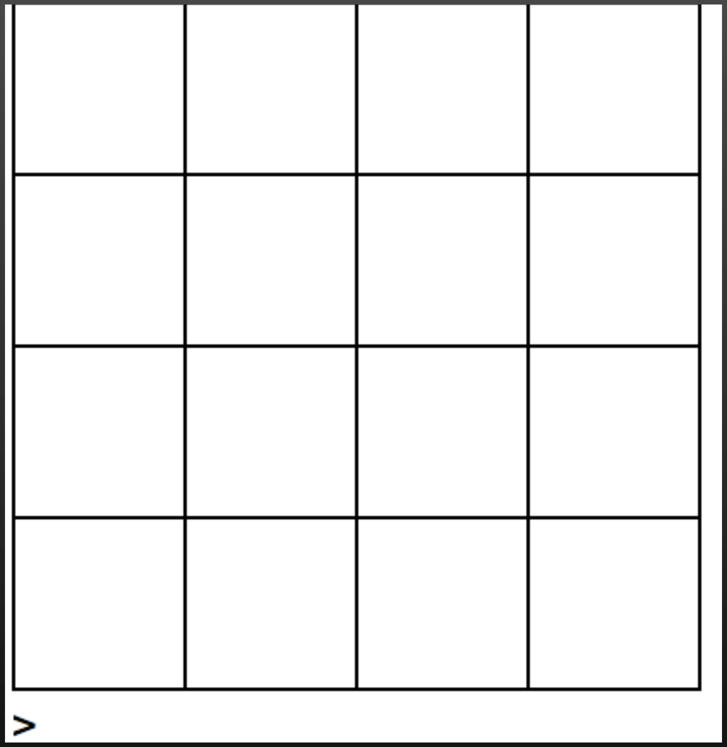
Weren’t computers supposed to be **labor-saving devices**?

```
(above (beside (square 50 "outline" "black")
               (square 50 "outline" "black")
               (square 50 "outline" "black")
               (square 50 "outline" "black"))
       (beside (square 50 "outline" "black")
               (square 50 "outline" "black")
               (square 50 "outline" "black")
               (square 50 "outline" "black"))
       (beside (square 50 "outline" "black")
               (square 50 "outline" "black")
               (square 50 "outline" "black")
               (square 50 "outline" "black"))
       (beside (square 50 "outline" "black")
               (square 50 "outline" "black")
               (square 50 "outline" "black")
               (square 50 "outline" "black")))
```

Simplifying with names

```
(define unit (square 50 "outline"
    "black"))
```

```
(above (beside unit unit unit unit)
       (beside unit unit unit unit)
       (beside unit unit unit unit)
       (beside unit unit unit unit))
```



Defining new names

(define *name value*)

- Tells system that *name* now refers to *value*
  - *Name* must be a valid variable name
  - But *value* can be an arbitrary expression
- Has to be executed to take effect
- **Naming** is the most basic **abstraction** mechanism

define isn’t a procedure

Why?

Rules of computation in Racket

- If it's a **constant** (a number or string)
  - It's its own value
  - Return it
- If it's a **variable name** (e.g. a word, hyphenated-phrase, or e.g. symbol)
  - Look up its value in the **dictionary**
  - Return (output) it
- If it has **parens** (i.e. it looks like "(*a b c* ...)")
  - Find the values of *a*, *b*, *c*, etc. using these same rules
  - The value of *a* had **better be a procedure**
  - Call it with the values of *b*, *c*, etc. as inputs
  - Return its output

Special forms

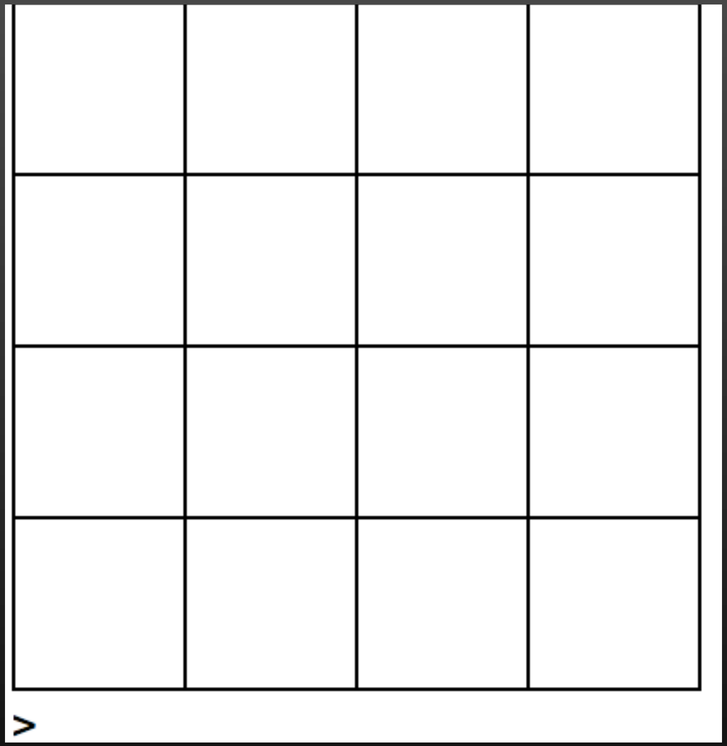
- Define is a special case that **works differently** from procedure calls
  - For a procedure call you always **replace input expressions** with their values
    - (+ (+5 3) 2)
  - But if you're defining a variable, it **doesn't have a value yet**!
    - (define a 5)
    - What is the value of a???

- There are a few special kinds of expressions like this, called **special forms**
  - We'll learn a few more, but not very many

This is still kind of a bad way to make a grid

```
(define unit (square 50 "outline"
    "black"))

(above (beside unit unit unit unit)
          (beside unit unit unit unit)
          (beside unit unit unit unit)
          (beside unit unit unit unit))
```



How can we **do better**?

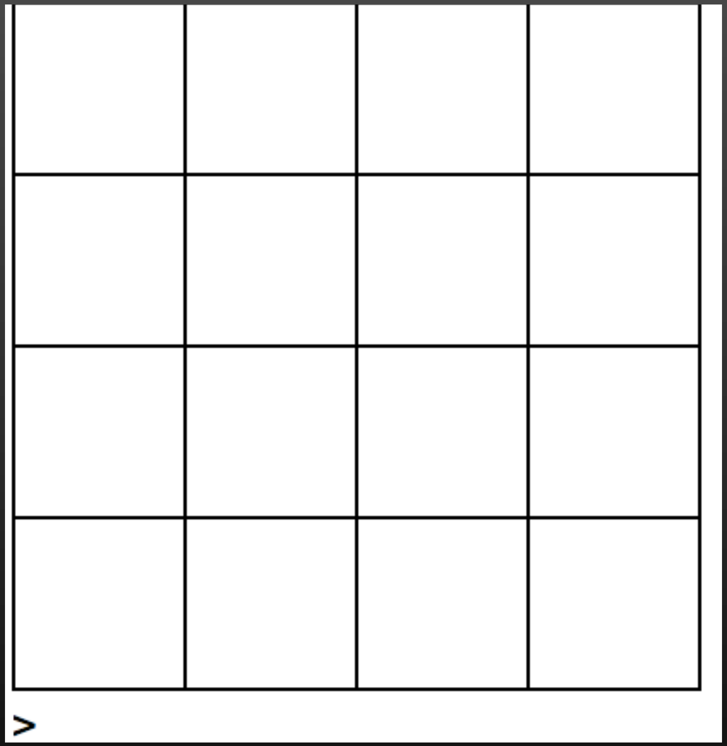
(Don’t say “loop”; we won’t get to those until next time)

Name the row

```
(define unit (square 50 "outline"
    "black"))

(define row (beside unit unit unit unit))

(above row row row row)
```



Your client called

Now they also want one with **circles** rather than squares

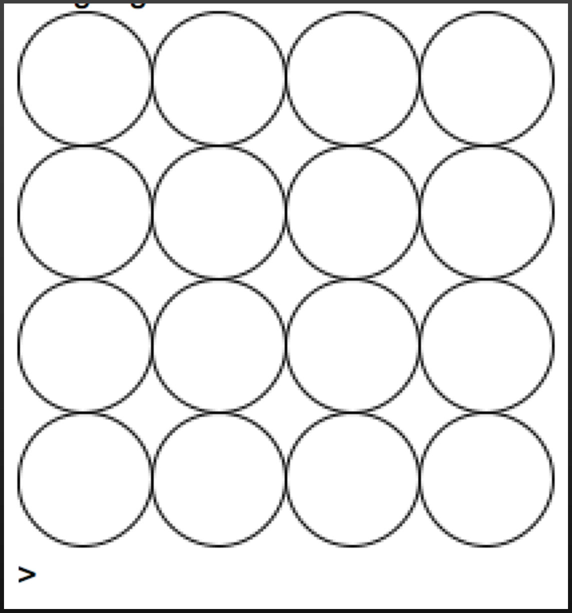
New and improved

```
(require 2htdp/image)

(define circ-unit
   (circle 25
           "outline"
           "black"))

(define circ-row
   (beside circ-unit circ-unit
           circ-unit circ-unit))

(above circ-row circ-row
       circ-row circ-row)
```



Your client called

Now they want a **grid of grids** of circles…

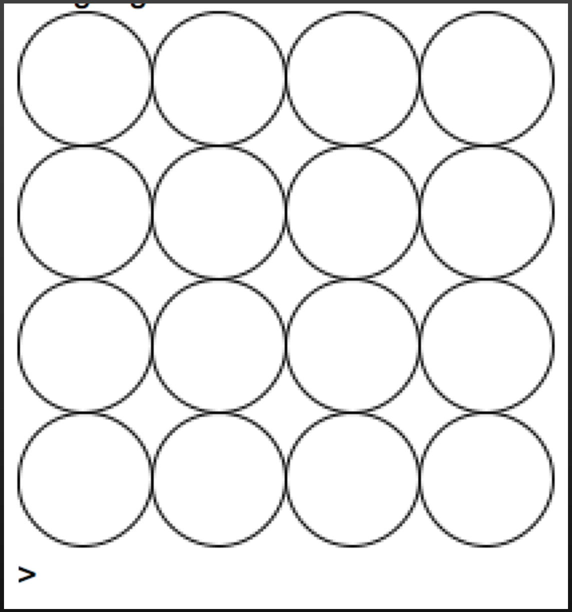
What’s wrong with this?

- Defining unit and row saves us a little bit of work, but …

- What we really want to do is to **name the pattern** of making a grids of shapes…
- We need **abstraction**

What we want to be able to say

```
(grid (circle 25 “outline” “black”))
```



```
>
```

What kind of a thing is grid?

(grid (circle 25 “outline” “black”))

- Well, it takes **inputs**
- It returns an **output**
- It must be a **procedure**…

- In fact, we even know what it should do
  - It should **make a row** from its argument
  - Then **make a stack** of the rows
  - And **return** it

Compound procedures

(lambda (*arg_1 arg_2 … arg_n*) *exp*)
(λ (*arg_1 arg_2 … arg_n*) *exp*)

Procedures are **just another data object**

- You can construct new procedures using **λ expressions**
- When called, the procedure
  - Sets the local names *arg_1 arg_2 … arg_n* to the arguments passed to the procedure
  - Computes the value of exp using the values of the arguments
  - Returns the value of *exp*

- Note: you **type the λ symbol**
  - by first typing command-\ on macs or control-\ on Windows
  - But you can also just type lambda