

MAU11601

Lecture 04

2026-09-30

# Shapes

We met `rectangle` and `circle` in the eclipse example. Some other available shapes are `ellipse`, `wedge`, `line`, `triangle`, `square` and `star`.

Guessing the types and order of the arguments can be tricky, but there are some patterns, e.g. horizontal precedes vertical if appropriate, the last argument is a colour, the second to last is usually `'solid` or `'outline`, etc.

When you write a lot of related functions you should decide on a set of conventions like this.

You can't remember everything. That's why (reference) documentation exists. In this case the documentation for `2htdp/image`, at

<https://docs.racket-lang.org/teachpack/2htdpimage.html>.

Part of your project is writing proper documentation for your programme.

The most common shapes are predefined, but it's impossible to anticipate all needs so there is a `polygon` function.

It takes a list of `posns` as an argument, so we need to talk about lists.

# Text

If you want text to appear in the interaction window you can evaluate an expression of string type, but if you want it to appear on the screen you need to convert it to an image and place it on a scene.

The `text` function does this, like

```
(place-image (text "I am a mouse!" 12 "purple") ... ).
```

What if you want to display a number, like a counter?

You first need to convert the number to a string, using the `number->string` function, then to an image, as above.

# Lists

I said polygon wants a list of posns, but what is a list and how do we make one?

We can create an empty list with `empty` or `null` or `'()` or `(list)`.

We can use the `list` function to create a list with its arguments as items, e.g.

`(list 2 3 5 7 11)` for the list of the first five primes.

If you type this in the interaction window in BSL you get

```
(cons 2 (cons 3 (cons 5 (cons 7 (cons 11 '())))))
```

What's that?

The `'()` is the empty list. `cons` is a function which takes an item and a list and creates a new list with that item added at the start.

So `(cons 11 '())` is a list with just 11 as an item, `(cons 7 (cons 11 '()))` is a list with 7 and 11 as items, etc.

`cons` adds items to the start of the list, so we add them in reverse order.

We write functions with prefix notation, so they are written from last applied to first.

`(cons 2 (cons 3 (cons 5 ... lists them in the right order because two dumb decisions cancel out!`

## Infix notation for lists

The `infix.rkt` module I wrote and put on Codeberg recognises `::` as an infix notation for `cons`, so you can write `{2 :: 3 :: 5 :: 7 :: 11 :: null}` instead of `(cons 2 (cons 3 (cons 5 (cons 7 (cons 11 '())))))`.

Unlike `+` and `-`, `::` is right associative, so the `::`s will be evaluated from right to left, as you'd want.

`::` has lower precedence than the arithmetic operations, so

`{1 + 1 :: 6 / 2 :: 8 - 3 :: 7 * 1 :: 11 :: null}` doesn't need parentheses.

## More lists

The Beginning Student Language has lists, but it displays them in an annoying way, so we'll now switch to the Intermediate Student Language, which displays

```
(list 2 3 5 7 11) as (list 2 3 5 7 11).
```

The internal representation is the same, only the display is different.

The ISL also introduces new and useful list functions like `map`, which takes a function and a list and gives a new list with that function applied to each item of the old list.

For example, with `(define (sq x) (* x x))` as before,

```
(map sq (list 2 3 5 7 11)) returns (list 4 9 25 49 121).
```

Another useful thing you can do with lists is `filter` them. This takes a predicate and a list and gives you a list with only those items satisfying the predicate, so

```
(filter odd? (list 2 3 5 7 11)) gives (list 3 5 7 11).
```

You can also apply a function of a variable number of arguments to a list, like `(apply + (list 2 3 5 7 11))`, which gives 28, i.e. what you would get from `(+ 2 3 5 7 11)`.

Question: what should `(apply and (list #false #true #false))` evaluate to?

## andmap and ormap

Often you want to test every item in a list with a predicate and check whether all of them are true, or at least one is true.

The obvious way to do this is `(apply and (map predicate list))` or `(apply or (map predicate list))`, but this doesn't work because `and` and `or` are not functions.

For this reason Racket provides functions `andmap` and `ormap`.

Like `and` and `or`, they stop once the value is determined, i.e. once they've seen a `#false` in the case of `and` or a `#true` in the case of `or`.

Suppose, for example, you have a maze, stored as a list of walls (boxes), and you want to know whether a move would put you inside a wall.

Define a function to check whether the new position is in a box, and then use `ormap` to check whether it's in any box in the list.

# Folds

`map` applies a function to each item in the list. These function applications don't interact in any way. We could run them all in parallel.

Sometimes you want something more complicated, where you keep track of intermediate results. Suppose, for example, you want to place copies of an image at a list of positions on a scene. It's unclear how to do this with `map` and `place-image`. We want to go through the list, producing a new scene at each stage, and then keep only the last. Suppose `initial` is the initial scene, `posns` is a list of positions, and `image` is the image.

```
(define (place-one p s) (place-image image (posn-x p) (posn-y p) s))
```

is then a function which places an image at `p` on `s`, returning a new scene.

We'd like to apply this function repeatedly, taking `p` to be the items in `posns` and `s` to be the result of the previous placement.

Then `(foldl place-image initial 1)` does what you want. To process the list in reverse order use `foldr` instead.

## Fold examples

In general `(foldl f s (list i_1 i_2 ... i_n))` returns `(f i_n ( ... (f i_2 (f i_1 s))))` while `(foldr f s (list i_1 i_2 ... i_n))` returns `(f i_1 (f i_2 (... (f i_n s))))`.

What is `(foldl + 0 (list 2 3 5 7 11))`?

The intermediate results are 0, 2, 5, 10, 17 and 28, so the answer is 28.

What is `(foldr + 0 (list 2 3 5 7 11))`?

Still 28, but computed differently. This time the intermediate results are 0, 11, 18, 23, 23 and 28.

What is `(foldl cons empty (list 2 3 5 7 11))`?  
`(list 11 7 5 3 2)`.

What is `(foldr cons empty (list 2 3 5 7 11))`?  
`(list 2 3 5 7 11)`.

# Advent of Code

If you want practice coding, Advent of Code, which runs every year, is a good resource. Every December there are 12-25 two-part programming puzzles. You have to solve the first part to unlock the second part.

The second part is always related to the first part. Usually if you solve the first part in sensible way you can modify your first part answer, otherwise you have to start from scratch.

It's language agnostic. You are given puzzle input and need to produce output, but you can use whatever language you want to produce it. The output is checked, not the code.

The puzzles are artificial, but well structured to test (and maybe teach) useful skills.

It's been running for a decade, so there are hundreds of puzzles in the archive.

Tomorrow we'll look at <https://adventofcode.com/2025/day/1>.