

MAU11601

Lecture 01

2026-09-23

Definitions and mutability

Definitions assign names to values, in a particular context.

The same value can have many different names in a particular context, but there can only be one value for a given name. Redefinition isn't allowed.

So `(define four 4)` `(define ceathair 4)` works but
`(define many 4)` `(define many 5)` doesn't.

That's a design choice. Some languages let you do that; some don't.

When writing programmes you may think you want a language which lets you do whatever you want. When debugging them you may wish it didn't.

Note that pretty much all languages, including Racket, allow `(define five 4)`.

Names usually signify what the programmer intended. They don't tell you what the programmer has really done.

Trusting names (and comments) is a common source of bugs!

Comments

Racket, like almost all programming languages, allows comments. Everything after a `;` is a comment, and does not affect evaluation of expressions.

Comments are often used to comment on the code, to improve readability, for yourself or others.

Like names, comments tell you about intent, not reality.

```
(define (av x) (if (> 0 x) x (- x))) ; absolute value
```

is not the absolute value function!

Descriptive names and lots of comments are often described as good style. But the compiler (or interpreter) doesn't verify their accuracy, so treat them with caution.

Comments are also useful for code you don't want evaluated, but which you don't want to erase yet.

You can comment out a section by putting `;`'s at the start of each line but DrRacket also lets you select a section and comment out everything in it.

Primitive data types

Languages generally provide automatic support for some data types, e.g. integers, strings, etc., and allow you to build more complicate structures from those. Often someone has already done that and packaged it in a library you can use.

Which data types are primitive varies from language to language. E.g. Fortran and Racket have complex numbers as primitive data types. C (mostly) doesn't. Racket has boolean values. Fortran and C don't. C and Fortran distinguish between single and double precision floating point numbers, while Racket doesn't.

One nice, and unusual, feature of Racket is that images are a primitive data type. You can copy an image, paste it directly into your code, define a variable to refer to it, etc. Often you want to build up a composite image from individual images. You might have a fixed background and want to move some figures on it, for example.

If you haven't already, you'll need to install the universe and image teachpacks. To use them, put `(require 2htdp/image)` and `(require 2htdp/universe)` just under the `#lang` line.

Examples, design philosophy

A scene is built from from an empty scene by placing images on it.

`(empty-scene 600 400)` creates an empty scene 600 pixels wide and 400 pixels high.

Assuming `cat` is the name we've defined for the image of a cat,

`(place-image cat 200 100 (empty-scene 600 400))` places that image at position `(200,100)` on that scene.

We can create an interactive environment as follows:

- Come up with a datatype describing the state of our world
- Give an instance of that datatype describing the initial state of the world.
- Define functions describing how the state changes in response to keyboard input, mouse events, the passage of time, etc.
- Define a function which creates a scene from the state of the world
- Hand all of that off to some mechanism which simulates the evolution of our world.

The `universe` teachpack provides such a mechanism, named `big-bang`.

Example: eclipse simulator

If you missed the eclipse in August we now have enough tools to simulate one.

- The (relevant part of the) state of the world is a single integer, the number of seconds to totality.
- We choose some number as the initial time to totality.
- Eclipses ignore keyboard input and mouse events, so we just need to describe the passage of time. The relevant function, which should be applied every second, just decrements the time to totality.
- The drawing function just draws a background and then two circles of the same radius, with centres offset in proportion to the time remaining.

We'll define `start-time`, `update` and `draw` in the definitions window and then type

```
(big-bang start-time (name "eclipse")  
            (to-draw draw) (on-tick update))
```

in the interaction window.

Simulator, implementation

We'll start time before the eclipse. The eclipse starts about an hour before totality, so `(define start-time 4500)` should be fine.

Remember that we're measuring time in seconds. Racket, like most programming languages, doesn't keep track of units, so you have to.

The update function takes the state of the world, i.e. time to totality, and gives you the new state on clock tick later. So if the clock ticks once per second then

`(define (update n) (- n 1))` should work.

The draw function is not much harder. We'll build it up one piece at a time. First we decide on the width and height of our scene and the radius of our circles:

`(define width 1000) (define height 1000) (define radius 360).`

The sun will have a fixed centre: `(define sun-x (/ width 2))`

`(define sun-y (/ height 2)).`

In the infix notation this is `(define sun-x {width / 2})`

`(define sun-y {height / 2})`

Simulator, implementation, continued

Now we define some images:

```
(define sky (rectangle width height 'solid 'lightblue))  
(define sun (circle radius 'solid 'yellow))  
(define moon (circle radius 'solid 'white)).
```

We can combine the parts which don't change into a fixed background scene:

```
(define background (place-image sun sun-x sun-y  
  (place-image sky sun-x sun-y (empty-scene width height))))
```

We need a function which calculates the vertical position of the moon from the time to totality: (define (moon-y s) (+ sun-y (/ (* s radius) 1800))).

Here the infix notation is noticeably more readable:

```
(define (moon-y s) {sun-y + s * radius / 1800})
```

Simulator, implementation, concluded

And finally we can write our draw function:

```
(define (draw s) (place-image moon sun-x (moon-y s) background)).
```

Then typing

```
(big-bang start-time (name "eclipse")  
  (to-draw draw) (on-tick update))
```

in the interaction window should then show us an eclipse.

It goes faster than a real eclipse because the default clock speed is 28 updates per second.

This is the general pattern for animations. Here the movement is determined by clock ticks but the pattern is similar for movement controlled by the keyboard or mouse.