

MAU11601

Lecture 03

2026-09-29

Types of program, termination

Most introductory programming modules use a batch programming model. A program take some input (all at once), produces some output, and then quits.

Some programmes naturally work this way, but not all.

For a batch programme, guaranteed termination is good. You want it to end.

The programmes we've looked at are interactive. Termination is not necessarily good.

Do you want your database server to die?

You might still want termination in some situations though, e.g. on non-recoverable errors, when the user clicks a quit button, etc.

Just as `big-bang` can take `on-tick`, `on-key` and `to-draw` parameters, it can also take a `stop-when` parameter, a predicate which takes a world state and determines whether to stop.

We can define such a predicate and hand it to `big-bang` with the rest of its arguments.

To quit in response to keyboard input, add a boolean `quit?` to your world structure and use `world-quit?` as your `stop-when` parameter.

Booleans

We've used boolean expressions, but not discussed them in detail.

We can use booleans in various ways:

- in `cond` statements
- in `if` statements
- as predicates, like the `stop-when` parameter

You can simulate `if` with `cond` and vice versa. `(if x y z)` is the same as `(cond ([x y] [else z]))` and `(cond [a x] [b y] [else z])` is the same as `(if a x (if b y z))`. We only have two different conditionals to help readability, not to provide power.

`big-bang` internally uses its `stop-when` parameter in an `if` expression, so really there's only one way to use booleans.

To express more complicated conditions we combine booleans with the boolean operators `and`, `or`, and `not`.

We can define a structure for intervals: `(define-struct interval (left right))`.

Booleans, continued

Although not enforced, the intent is that `left` and `right` are numbers, so this can be thought of as an ordered pair of numbers. A `posn` is also an ordered pair of numbers, but an `interval` isn't a `posn` and a `posn` isn't an `interval`. This is enforced.

It would still be enforced even if we'd named the fields `x` and `y`.

We can define a function to check whether a number lies in an (open) interval:

```
(define in? (x I)
  (and (> x (interval-left I)) (< x (interval-right I))))
```

So `(in? 4 (make-interval 3 5))` returns `#true` while

`(in? 2 (make-interval 3 5))` returns `#false`.

In addition to the boolean operator `and` we also have `or` and `not`.

If you want to express, for example, the condition for a position in a maze to be in a wall, this is quite useful.

What are `and` and `or`. Are they functions? This is where things get tricky.

Are boolean operators functions? (1/3)

Mathematically, a function has a well defined domain and, for each member of that domain there is one and only one value of the function.

So `not` is a function from booleans to booleans and `and` and `or` are functions from pairs of booleans to booleans.

In computing, the word function means something different, or really several things, all different.

For example, Racket, and BSL, has a function `random` to produce random numbers. `(random 7)` returns a random number from 0 to 6. This is very useful in creating a game, but this is not a function in the mathematical sense.

`+` is a function, in the mathematical and programming senses. To evaluate `(+ x y)` you need the values of both `x` and `y`. So if you give it nonsense for one of those, it should complain, and it does.

`(+ 1 (/ 2 0))` and `(+ 1 "three")` produce errors.

Are boolean operators functions? (2/3)

`and` is different. To evaluate `(and p q)` we might not need to evaluate both `p` and `q`. If `p` is `#false` then we don't need `q`. Should we evaluate it anyway?

Should `(and (= 0 1) (= 2 (/ 3 0)))` evaluate to `#false` or throw an error? What about `(and (= 0 1) "wombat")`?

What should happen isn't obvious, but what does happen, in Racket, is that both produce `#false`.

Pretty much all computer languages agree that the first should behave this way. Many, but not all, will object to the second.

What about `(and (= 2 (/ 3 0)) (= 0 1))`? In Racket, and almost all languages, this gives a division by zero error, so `(and p q)` isn't the same as `(and q p)`.

What should functions do with arguments which aren't needed? Should they be evaluated?

This question doesn't have an obvious answer and different languages behave differently.

What does Racket do?

Are boolean operators functions? (3/3)

Does Racket (and BSL) evaluate unused arguments to functions?

Let's write a function which doesn't use some of its arguments.

```
(define (middle x y z) y)
```

Then `(middle 2 3 4)` returns 3, as it should. What about `(middle 2 3 (/ 1 0))`?

It gives a division by zero argument, so the unused argument is still evaluated.

Evaluating arguments only when needed is called call by need or lazy evaluation.

Evaluating arguments regardless is called call by value or eager evaluation.

Most languages, including Racket, are eager. A few, like Haskell, are lazy.

In a lazy language `and` and `or` can be functions, but in an eager language they can't be.

`not` is fine. Its argument is always needed immediately, so lazy and eager evaluation behave the same.

If `and` and `or` aren't functions, what are they?

In Racket, `(and p q)` is secretly rewritten to `(if p q #false)` and `(or p q)` is secretly rewritten to `(if p #true q)`.

`if` can't be a function either, for similar reasons. It's called a special form.

Mouse input

Just as we can tell `big-bang` what to do on clock ticks or keyboard input, we can tell it how to respond to mouse movements.

We just give it a parameter `on-mouse`, a function which takes as arguments the world state, the horizontal and vertical position, and the type of event, and returns the updated world state.

The types of events are `button-down`, `button-up`, `drag`, `move`, `enter` and `leave`.

Just like you can test key matches with the `key=?` function, you can test mouse events with the `mouse=?` function.

Usually your `on-mouse` function will have a big `cond` expression for the different event types.

`button-down` and `button-up` refer to the mouse button. `enter` and `leave` refer to the view area, i.e. the scene.

`move` and `drag` are both mouse movements. A movement is a drag if the button is held down and a move if not.

Note that mouse positions are reported periodically, not continuously.