Teachpacks

Version 4.1.5

March 21, 2009

Teaching languages are small subsets of a full programming language. While such restrictions simplify error diagnosis and the construction of tools, they also make it impossible (or at least difficult) to write some interesting programs. To circumvent this restriction, it is possible to import teachpacks into programs written in a teaching language.

In principle, a teachpack is just a library written in the full language, not the teaching subset. Like any other library, it may export values, functions, etc. In contrast to an ordinary library, however, a teachpack must enforce the contracts of the ”lowest” teaching language into which it is imported and signal errors in a way with which students are familiar at that level.

This chapter covers the teachpacks for *How to Design Programs* and *How to Design Classes*.

Contents

1 HtDP Teachpacks

1.1 Manipulating Images: `"image.ss"`

```
(require htdp/image)
```

The teachpack provides primitives for constructing and manipulating images. Basic, colored images are created as outlines or solid shapes. Additional primitives allow for the composition of images.

1.1.1 Images

```
(image? x) → boolean?
  x : any/c
```

Is `x` an image?

1.1.2 Modes and Colors

Mode `(one-of/c 'solid 'outline "solid" "outline")`

A Mode is used to specify whether painting a shape fills or outlines the form.

```
(struct color (red green blue))
  red : (and/c natural-number/c (<=/c 255))
  green : (and/c natural-number/c (<=/c 255))
  blue : (and/c natural-number/c (<=/c 255))
```

RGB `color?`

A RGB describes a color via a shade of red, blue, and green colors (e.g., `(make-color 100 200 30)`).

Color `(or/c symbol? string? color?)`

A Color is a color-symbol (e.g., `'blue`) or a color-string (e.g., `"blue"`) or an RGB structure.

```
(image-color? x) → boolean?
  x : any
```

Determines if the input is a valid image Color.

1.1.3 Creating Basic Shapes

In DrScheme, you can insert images from your file system. Use PNG images instead whenever possible for insertions. In addition, you can create basic shapes with the following functions.

```
(rectangle w h m c) → image?
  w : (and/c number? (or/c zero? positive?))
  h : (and/c number? (or/c zero? positive?))
  m : Mode
  c : Color
```

Creates a `w` by `h` rectangle, filled in according to `m` and painted in color `c`

```
(circle r m c) → image?
  r : (and/c number? (or/c zero? positive?))
  m : Mode
  c : Color
```

Creates a circle or disk of radius `r`, filled in according to `m` and painted in color `c`

```
(ellipse w h m c) → image?
  w : (and/c number? (or/c zero? positive?))
  h : (and/c number? (or/c zero? positive?))
  m : Mode
  c : Color
```

Creates a `w` by `h` ellipse, filled in according to `m` and painted in color `c`

```
(triangle s m c) → image?
  s : number?
  m : Mode
  c : Color
```

Creates an upward pointing equilateral triangle whose side is `s` pixels long, filled in according to `m` and painted in color `c`

```
(star n outer inner m c) → image?
  n : (and/c number? (>=/c 2))
  outer : (and/c number? (>=/c 1))
  inner : (and/c number? (>=/c 1))
  m : Mode
  c : Color
```

Creates a multi-pointed star with `n` points, an `outer` radius for the max distance of the points to the center, and an `inner` radius for the min distance to the center.

```
(regular-polygon s r m c [angle]) → image?
  s : side
  r : number?
  m : Mode
  c : Color
  angle : real? = 0
```

Creates a regular polygon with `s` sides inscribed in a circle of radius `r`, using mode `m` and color `c`. If an angle is specified, the polygon is rotated by that angle.

```
(line x y c) → image?
  x : number?
  y : number?
  c : Color
```

Creates a line colored `c` from (0,0) to `(x ,y)`. See `add-line` below.

```
(text s f c) → Image
  s : string?
  f : (and/c number? positive?)
  c : Color
```

Creates an image of the text `s` at point size `f` and painted in color `c`.

1.1.4 Basic Image Properties

To understand how images are manipulated, you need to understand the basic properties of images.

```
(image-width i) → integer?
  i : image?
```

Obtain `i`'s width in pixels

```
(image-height i) → integer?
  i : image?
```

Obtain `i`'s height in pixels

For the composition of images, you must know about *pinholes*. Each image, including primitive ones, come with a pinhole. For images created with the above primitives, the pinhole is at the center of the shape except for those created from `line` and `text`. The `text` function puts the pinhole at the upper left corner of the image, and `line` puts the pinhole at the beginning of the line (meaning that if the first two arguments to `line` are positive, the pinhole is also in the upper left corner). The pinhole can be moved, of course, and compositions locate pinholes according to their own rules. When in doubt you can always find out where the pinhole is and place it where convenient.

```
(pinhole-x i) → integer?
  i : image?
```

Determines the `x` coordinate of the pinhole, measuring from the left of the image.

```
(pinhole-y i) → integer?
  i : image?
```

Determines the `y` coordinate of the pinhole, measuring from the top (down) of the image.

```
(put-pinhole i x y) → image?
  i : image?
  x : number?
  y : number?
```

Creates a new image with the pinhole in the location specified by `x` and `y`, counting from the left and top (down), respectively.

```
(move-pinhole i delta-x delta-y) → image?
  i : image?
  delta-x : number?
  delta-y : number?
```

Creates a new image with the pinhole moved down and right by `delta-x` and `delta-y` with respect to its current location. Use negative numbers to move it up or left.

1.1.5 Composing Images

Images can be composed, and images can be found within compositions.

```
(add-line i x y z u c) → image?
  i : image?
```

```
x : number?
y : number?
z : number?
u : number?
c : Color
```

Creates an image by adding a line (colored `c`) from `(x ,y)` to `(z ,u)` to image `i`.

```
(overlay img img2 img* ...) → image?
  img : image?
  img2 : image?
  img* : image?
```

Creates an image by overlaying all images on their pinholes. The pinhole of the resulting image is the same place as the pinhole in the first image.

```
(overlay/xy img delta-x delta-y other) → image?
  img : image?
  delta-x : number?
  delta-y : number?
  other : image?
```

Creates an image by adding the pixels of `other` to `img`.

Instead of lining the two images up on their pinholes, `other`'s pinhole is lined up on the point:

```
(make-posn (+ (pinhole-x img) delta-x)
           (+ (pinhole-y img) delta-y))
```

The pinhole of the resulting image is the same place as the pinhole in the first image.

The same effect can be had by combining `move-pinhole` and `overlay`,

```
(overlay img
         (move-pinhole other
                       (- delta-x)
                       (- delta-y)))
```

```
(image-inside? img other) → boolean?
  img : image?
  other : image?
```

Determines whether the pixels of the second image appear in the first.

Be careful when using this function with jpeg images. If you use an image-editing program to crop a jpeg image and then save it, `image-inside?` does not recognize the cropped image, due to standard compression applied to JPEG images.

```
(find-image img other) → posn?
  img : image?
  other : image?
```

Determines where the pixels of the second image appear in the first, with respect to the pinhole of the first image. If `(image-inside? img other)` isn't true, `find-image` signals an error.

1.1.6 Manipulating Images

Images can also be shrunk. These "shrink" functions trim an image by eliminating extraneous pixels.

```
(shrink-tl img width height) → image?
  img : image?
  width : number?
  height : number?
```

Shrinks the image to a `width` by `height` image, starting from the *top-left* corner. The pinhole of the resulting image is in the center of the image.

```
(shrink-tr img width height) → image?
  img : image?
  width : number?
  height : number?
```

Shrinks the image to a `width` by `height` image, starting from the *top-right* corner. The pinhole of the resulting image is in the center of the image.

```
(shrink-bl img width height) → image?
  img : image?
  width : number?
  height : number?
```

Shrinks the image to a `width` by `height` image, starting from the *bottom-left* corner. The pinhole of the resulting image is in the center of the image.

```
(shrink-br img width height) → image?
  img : image?
  width : number?
  height : number?
```

Shrinks the image to a `width` by `height` image, starting from the *bottom-right* corner. The pinhole of the resulting image is in the center of the image.

```
(shrink img left above right below) → image?
  img : image?
  left : number?
  above : number?
  right : number?
  below : number?
```

Shrinks an image around its pinhole. The numbers are the pixels to save to left, above, to the right, and below the pinhole, respectively. The pixel directly on the pinhole is always saved.

1.1.7 Miscellaneous Image Manipulation and Creation

The last group of functions extracts the constituent colors from an image and converts a list of colors into an image.

```
List-of-color : list?
```

is one of:

```
; -- empty
; -- (cons Color List-of-color)
; Interpretation: represents a list of colors.
```

```
(image->color-list img) → List-of-color
  img : image?
```

Converts an image to a list of colors.

```
(color-list->image l width height x y) → image?
  l : List-of-color
  width : natural-number/c
  height : natural-number/c
  x : natural-number/c
  y : natural-number/c
```

Converts a list of colors `l` to an image with the given *width* and *height* and pinhole (*x*,*y*) coordinates, specified with respect to the top-left of the image.

The remaining functions provide alpha-channel information as well. Alpha channels are a measure of transparency; 0 indicates fully opaque and 255 indicates fully transparent.

```
(struct alpha-color (alpha red green blue))
  alpha : (and/c natural-number/c (<=/c 255))
  red : (and/c natural-number/c (<=/c 255))
  green : (and/c natural-number/c (<=/c 255))
  blue : (and/c natural-number/c (<=/c 255))
```

A structure representing an alpha color.

```
(image->alpha-color-list img) → (list-of alpha-color?)
  img : image?
```

to convert an image to a list of alpha colors

```
(alpha-color-list->image l width height x y) → image?
  l : (list-of alpha-color?)
  width : integer?
  height : integer?
  x : integer?
  y : integer?
```

Converts a list of `alpha-colors` `l` to an image with the given *width* and *height* and pinhole (*x*,*y*) coordinates, specified with respect to the top-left of the image.

1.2 Simulations and Animations: "world.ss"

```
(require htdp/world)
```

Note: For a quick and educational introduction to the teachpack, see How to Design Programs, Second Edition: Prologue. As of August 2008, we also have a series of projects available as a small booklet on How to Design Worlds.

The purpose of this documentation is to give experienced Schemers a concise overview for using the library and for incorporating it elsewhere. The last section presents §1.2.5 “A First Example” for an extremely simple domain and is suited for a novice who knows how to design conditional functions for symbols.

The teachpack provides two sets of tools. The first allows students to create and display a series of animated scenes, i.e., a simulation. The second one generalizes the first by adding interactive GUI features.

1.2.1 Basics

The teachpack assumes working knowledge of the basic image manipulation primitives and introduces a special kind of image: a scene.

Scene

```
(define (focus-at-0-0 i)
  (and (= (pinhole-x i) 0) (= (pinhole-y i) 0)))

(and/c image? focus-at-0-0)
```

The teachpack can display only Scenes, which are images whose pinholes are at position `(0 ,0)`.

`(empty-scene width height)` → Scene
  width : `natural-number/c`
  height : `natural-number/c`

Creates a *width* x *height* Scene.

`(place-image img x y s)` → Scene
  img : `image?`
  x : `number?`
  y : `number?`
  s : Scene

Creates a scene by placing *img* at `(x ,y)` into *s*; `(x ,y)` are comp. graph. coordinates, i.e., they count right and down from the upper-left corner.

1.2.2 Simple Simulations

`(run-simulation w h r create-image)` → `true`
  w : `natural-number/c`
  h : `natural-number/c`
  r : `number?`
  create-image : `(-> natural-number/c scene)`

creates and shows a canvas of width *`w`* and height *`h`* , starts a clock, making it tick every *`r`* (usually fractional) seconds. Every time the clock ticks, drscheme applies *`create-image`* to the number of ticks passed since this function call. The results of these applications are displayed in the canvas.

Example:

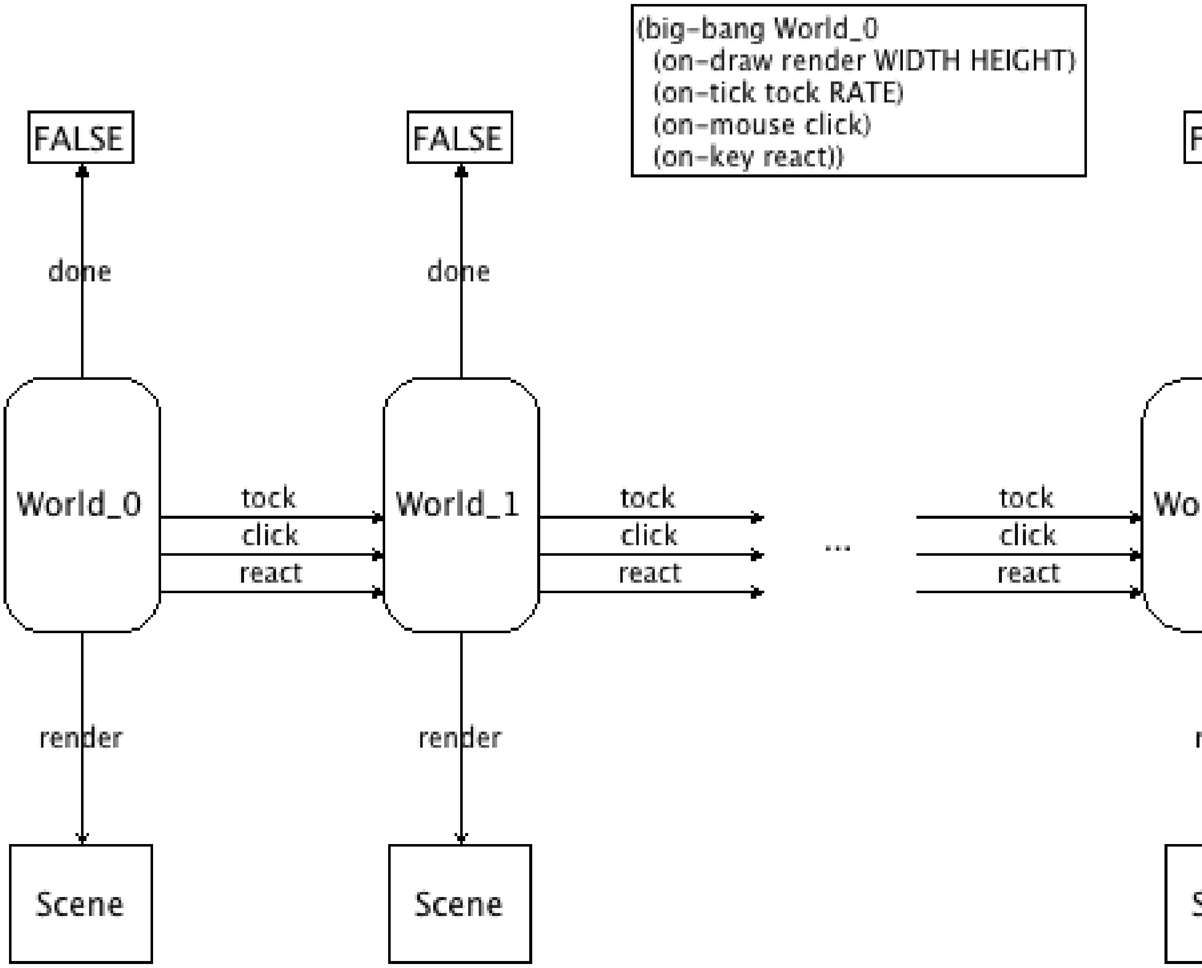
```
(define (create-UFO-scene height)
  (place-image UFO 50 height (empty-scene 100 100)))

(define UFO
  (overlay (circle 10 'solid 'green)
           (rectangle 40 4 'solid 'green)))

(run-simulation 100 100 (/ 1 28) create-UFO-scene)
```

1.2.3 Interactions

An animation starts from a given "world" and generates new ones in response to events on the computer. This teachpack keeps track of the "current world" and recognizes three kinds of events: clock ticks; keyboard presses and releases; and mouse movements, mouse clicks, etc.

Your program may deal with such events via the *installation* of *handlers*. The teachpack provides for the installation of three event handlers: `on-tick-event`, `on-key-event`, and `on-mouse-event`. In addition, it provides for the installation of a `draw` handler, which is called every time your program should visualize the current world.

The following picture provides an intuitive overview of the workings of ”world”.



```
(big-bang World_0
  (on-draw render WIDTH HEIGHT)
  (on-tick tock RATE)
  (on-mouse click)
  (on-key react))
```

The `big-bang` function installs *World_0* as the initial world; the callbacks *tock*, *react*, and *click* transform one world into another one; *done* checks each time whether the world is final; and *draw* renders each world as a scene.

World `any/c`

For animated worlds and games, using the teachpack requires that you provide a data definition for World. In principle, there are no constraints on this data definition. You can even keep it implicit, even if this violates the Design Recipe.

`(big-bang width height r world0) → true`

```
  width : natural-number/c
  height : natural-number/c
  r : number?
  world0 : World
(big-bang width height r world0 animated-gif?) → true
  width : natural-number/c
  height : natural-number/c
  r : number?
  world0 : World
  animated-gif? : boolean?
```

Creates and displays a `width` x `height` canvas, starts the clock, makes it tick every `r` seconds, and makes `world0` the current world. If it is called with five instead of four arguments and the last one (`animated-gif?`) is `true`, the teachpack allows the generation of images from the animation, including an animated GIF image.

```
(on-tick-event tock) → true
  tock : (-> World World)
```

Tell DrScheme to call `tock` on the current world every time the clock ticks. The result of the call becomes the current world.

KeyEvent `(or/c char? symbol?)`

A KeyEvent represents key board events, e.g., keys pressed or released, by the computer's user. A `char?` KeyEvent is used to signal that the user has hit an alphanumeric key. Symbols such as `'left`, `'right`, `'up`, `'down`, `'release` denote arrow keys or special events, such as releasing the key on the keypad.

```
(key-event? x) → boolean?
  x : any
```

is `x` a KeyEvent

```
(key=? x y) → boolean?
  x : key-event?
  y : key-event?
```

compares two KeyEvent for equality

```
(on-key-event change) → true
  change : (-> World key-event? World)
```

Tell DrScheme to call `change` on the current world and a KeyEvent for every keystroke the user of the computer makes. The result of the call becomes the current world.

Here is a typical key-event handler:

```
(define (change w a-key-event)
  (cond
    [(key=? a-key-event 'left)  (world-go w -DELTA)]
    [(key=? a-key-event 'right) (world-go w +DELTA)]
    [(char? a-key-event) w] ; to demonstrate order-free checking
    [(key=? a-key-event 'up)    (world-go w -DELTA)]
    [(key=? a-key-event 'down)  (world-go w +DELTA)]
    [else w]))
```

MouseEvent `(one-of/c 'button-down 'button-up 'drag 'move 'enter 'leave)`

A MouseEvent represents mouse events, e.g., mouse movements or mouse clicks, by the computer's user.

`(on-mouse-event clack)` → `true`
  `clack : (->` World `natural-number/c natural-number/c` MouseEvent World`)`

Tell DrScheme to call `clack` on the current world, the current `x` and `y` coordinates of the mouse, and and a MouseEvent for every action of the mouse by the user of the computer. The result of the call becomes the current world.

`(on-redraw to-scene)` → `true`
  `to-scene : (->` World Scene`)`

Tell DrScheme to call `to-scene` whenever the canvas must be redrawn. The canvas is usually re-drawn after a tick event, a keyboard event, or a mouse event has occurred. The generated scene is displayed in the world's canvas.

`(stop-when last-world?)` → `true`
  `last-world? : (->` World `boolean?)`

Tell DrScheme to call `last-world?` whenever the canvas is drawn. If this call produces `true`, the clock is stopped; no more tick events, KeyEvents, or MouseEvents are forwarded to the respective handlers. As a result, the canvas isn't updated either.

Example: The following examples shows that `(run-simulation 100 100 (/ 1 28) create-UFO-scene)` is a short-hand for three lines of code:

```
(define (create-UFO-scene height)
  (place-image UFO 50 height (empty-scene 100 100)))
```

```
(define UFO
  (overlay (circle 10 'solid 'green)
           (rectangle 40 4 'solid 'green)))

(big-bang 100 100 (/1 28) 0)
(on-tick-event add1)
(on-redraw create-UFO-scene)
```

Exercise: Add a condition for stopping the flight of the UFO when it reaches the bottom.

1.2.4 Scenes and Images

For the creation of scenes from the world, use the functions from §1.1 “Manipulating Images: `"image.ss"`”. The following two functions have turned out to be useful for the creation of scenes, too.

```
(nw:rectangle width height solid-or-filled c) → image?
  width : natural-number/c
  height : natural-number/c
  solid-or-filled : Mode
  c : Color
```

Creates a `width` x `height` rectangle, solid or outlined as specified by `solid-or-filled` and colored according to `c`, with a pinhole at the upper left corner.

```
(scene+line s x0 y0 x1 y1 c) → Scene
  s : Scene
  x0 : number?
  y0 : number?
  x1 : number?
  y1 : number?
  c : Color
```

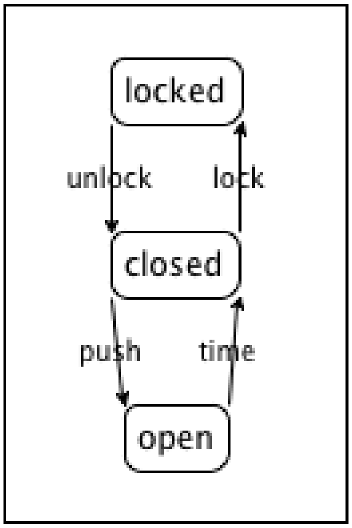
Creates a scene by placing a line of color `c` from `(x0 ,y0)` to `(x1 ,y1)` into `scene`; `(x ,y)` are comp. graph. coordinates; in contrast to the `add-line` function, this one cuts off those portions of the line that go beyond the boundaries of the given `s`.

1.2.5 A First Example

Understanding a Door

Say we want to represent a door with an automatic door closer. If this kind of door is locked, you can unlock it. While this doesn't open the door per se, it is now possible to do so. That is, an unlocked door is closed and pushing at the door opens it. Once you have passed through the door and you let go, the automatic door closer takes over and closes the door again. Of course, at this point you could lock it again.

Here is a picture that translates our words into a graphical representation:



The picture displays a so-called ”state machine”. The three circled words are the states that our informal description of the door identified: locked, closed (and unlocked), and open. The arrows specify how the door can go from one state into another. For example, when the door is open, the automatic door closer shuts the door as time passes. This transition is indicated by the arrow labeled ”time passes.” The other arrows represent transitions in a similar manner:

- ”push” means a person pushes the door open (and let's go);
- ”lock” refers to the act of inserting a key into the lock and turning it to the locked position; and
- ”unlock” is the opposite of ”lock”.

Simulations of the World

Simulating any dynamic behavior via a program demands two different activities. First, we must tease out those portions of our ”world” that change over time or in reaction to actions, and we must develop a data representation *D* for this information. Keep in mind that a good data definition makes it easy for readers to map data to information in the real world and vice versa. For all others aspects of the world, we use global constants, including graphical or visual constants that are used in conjunction with the rendering operations.

Second, we must translate the ”world” actions—the arrows in the above diagram—into interactions with the computer that the world teachpack can deal with. Once we have decided to use the passing of time for one aspect and mouse movements for another, we must develop functions that map the current state of the world—represented as data—into the next state of the world. Since the data definition D describes the class of data that represents the world, these functions have the following general contract and purpose statements:

```
; tick : D -> D
; deal with the passing of time
(define (tick w) ...)

; click : D Number Number MouseEvent -> D
; deal with a mouse click at (x,y) of kind me
; in the current world w
(define (click w x y me) ...)

; control : D KeyEvent -> D
; deal with a key event (symbol, char) ke
; in the current world w
(define (control w ke) ...)
```

That is, the contracts of the various hooks dictate what the contracts of these functions are once we have defined how to represent the world in data.

A typical program does not use all three of these actions and functions but often just one or two. Furthermore, the design of these functions provides only the top-level, initial design goal. It often demands the design of many auxiliary functions.

Simulating a Door: Data

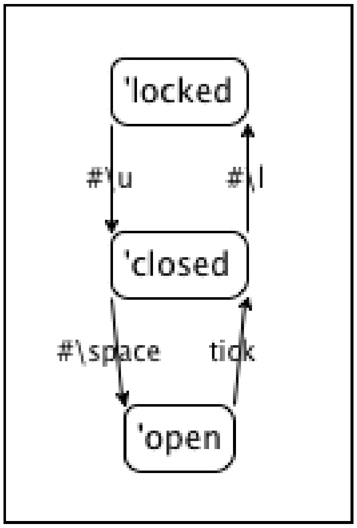
Our first and immediate goal is to represent the world as data. In this specific example, the world consists of our door and what changes about the door is whether it is locked, unlocked but closed, or open. We use three symbols to represent the three states:

SD

```
; DATA DEF.
; The state of the door (SD) is one of:
; -- 'locked
; -- 'closed
; -- 'open
```

Symbols are particularly well-suited here because they directly express the state of the door.

Now that we have a data definition, we must also decide which computer actions and interactions should model the various actions on the door. Our pictorial representation of the door's states and transitions, specifically the arrow from "open" to "closed" suggests the use of a function that simulates time. For the other three arrows, we could use either keyboard events or mouse clicks or both. Our solution uses three keystrokes: `#\u` for unlocking the door, `#\l` for locking it, and `#\space` for pushing it open. We can express these choices graphically by translating the above "state machine" from the world of information into the world of data:



Simulating a Door: Functions

Our analysis and data definition leaves us with three functions to design:

- `automatic-closer`, which closes the time during one tick;
- `door-actions`, which manipulates the time in response to pressing a key; and
- `render`, which translates the current state of the door into a visible scene.

Let’s start with `automatic-closer`. We know its contract and it is easy to refine the purpose statement, too:

```
; automatic-closer : SD -> SD
; closes an open door over the period of one tick
(define (automatic-closer state-of-door) ...)
```

Making up examples is trivial when the world can only be in one of three states:

given state	desired state
’locked	’locked
’closed	’closed
’open	’closed

```
; automatic-closer : SD -> SD
; closes an open door over the period of one tick

(check-expect (automatic-closer 'locked) 'locked)
(check-expect (automatic-closer 'closed) 'closed)
(check-expect (automatic-closer 'open) 'closed)

(define (automatic-closer state-of-door) ...)
```

The template step demands a conditional with three clauses:

```
(define (automatic-closer state-of-door)
  (cond
    [(symbol=? 'locked state-of-door) ...]
    [(symbol=? 'closed state-of-door) ...]
    [(symbol=? 'open state-of-door) ...]))
```

The examples basically dictate what the outcomes of the three cases must be:

```
(define (automatic-closer state-of-door)
  (cond
    [(symbol=? 'locked state-of-door) 'locked]
    [(symbol=? 'closed state-of-door) 'closed]
    [(symbol=? 'open state-of-door) 'closed]))
```

Don’t forget to run the example-tests.

For the remaining three arrows of the diagram, we design a function that reacts to the three chosen keyboard events. As mentioned, functions that deal with keyboard events consume both a world and a keyevent:

```
; door-actions : SD Keyevent -> SD
```

```
; key events simulate actions on the door
(define (door-actions s k) ...)
```

given state	given keyevent	desired state
’locked	#\u	’closed
’closed	#\l	’locked
’closed	#\space	’open
’open	—	’open

The examples combine what the above picture shows and the choices we made about mapping actions to keyboard events.

From here, it is straightforward to turn this into a complete design:

```
(define (door-actions s k)
  (cond
    [(and (symbol=? 'locked s) (key=? #\u k)) 'closed]
    [(and (symbol=? 'closed s) (key=? #\l k)) 'locked]
    [(and (symbol=? 'closed s) (key=? #\space k)) 'open]
    [else s]))

(check-expect (door-actions 'locked #\u) 'closed)
(check-expect (door-actions 'closed #\l) 'locked)
(check-expect (door-actions 'closed #\space) 'open)
(check-expect (door-actions 'open 'any) 'open)
(check-expect (door-actions 'closed 'any) 'closed)
```

Last but not least we need a function that renders the current state of the world as a scene. For simplicity, let’s just use a large enough text for this purpose:

```
; render : SD -> Scene
; translate the current state of the door into a large text
(define (render s)
  (text (symbol->string s) 40 'red))

(check-expecy (render 'closed) (text "closed" 40 'red))
```

The function `symbol->string` translates a symbol into a string, which is needed because `text` can deal only with the latter, not the former. A look into the language documentation revealed that this conversion function exists, and so we use it.

Once everything is properly designed, it is time to *run* the program. In the case of the world teachpack, this means we must specify which function takes care of tick events, key events, and redraws:

```
(big-bang 100 100 1 'locked)
```

```
(on-tick-event automatic-closer)
(on-key-event door-actions)
(on-redraw render)
```

Now it's time for you to collect the pieces and run them in DrScheme to see whether it all works.

1.3 Converting Temperatures: `"convert.ss"`

The teachpack `convert.ss` provides three functions for converting Fahrenheit temperatures to Celsius. It is useful for a single exercise in HtDP. Its purpose is to demonstrate the independence of "form" (user interface) and "function" (also known as "model").

```
(convert-gui convert) → true
  convert : (-> number? number?)
```

Consumes a conversion function from Fahrenheit to Celsius and creates a graphical user interface with two rulers, which users can use to convert temperatures according to the given temperature conversion function.

```
(convert-repl convert) → true
  convert : (-> number? number?)
```

Consumes a conversion function from Fahrenheit to Celsius and then starts a read-evaluate-print loop. The loop prompts users to enter a number and then converts the number according to the given temperature conversion function. A user can exit the loop by entering "x."

```
(convert-file in convert out) → true
  in : string?
  convert : (-> number? number?)
  out : string?
```

Consumes a file name *`in`*, a conversion function from Fahrenheit to Celsius, and a string *`out`*. The program then reads all the number from *`in`*, converts them according to *`convert`*, and prints the results to the newly created file *`out`*.

Warning: If *`out`* already exists, it is deleted.

Example: Create a file with name `"in.dat"` with some numbers in it, using your favorite text editor on your computer. Define a function `f2c` in the Definitions window and set teachpack to "convert.ss" and click RUN. Then evaluate

```
(convert-gui f2c)
```

```
; and
(convert-file "in.dat" f2c "out.dat")
; and
(convert-repl f2c)
```

Finally inspect the file `"out.dat"` and use the repl to check the answers.

1.4 Guessing Numbers: `"guess.ss"`

The teachpack provides operations to play a guess-the-number game. Each operation display a GUI in which a player can choose specific values for some number of digits and then check the guess. The more advanced operations ask students to implement more of the game.

```
(guess-with-gui check-guess) → true
  check-guess : (-> number? number? symbol?)
```

The *`check-guess`* function consumes two numbers: `guess`, which is the user's guess, and `target`, which is the randomly chosen number-to-be-guessed. The result is a symbol that reflects the relationship of the player's guess to the target.

```
(guess-with-gui-3 check-guess) → true
  check-guess : (-> digit? digit? digit? number? symbol?)
```

The *`check-guess`* function consumes three digits (`digit0`, `digit1`, `digit2`) and one number (`target`). The latter is the randomly chosen number-to-be-guessed; the three digits are the current guess. The result is a symbol that reflects the relationship of the player's guess (the digits converted to a number) to the target.

Note: `digit0` is the *least* significant digit that the user chose and `digit2` is the *most* significant one.

```
(guess-with-gui-list check-guess) → true
  check-guess : (-> (list-of digit?) number? symbol?)
```

The *`check-guess`* function consumes a list of digits (`digits`) and a number (`target`). The former is a list that makes up the user's guess, and the latter is the randomly chosen number-to-be-guessed. The result is a symbol that reflects the relationship of the player's guess (the digits converted to a number) to the target.

Note: the first item on `digits` is the *least* significant digit that the user chose, and the last one is the *most* significant digit.

1.5 MasterMinding: `"master.ss"`

The teachpack implements GUI for playing a simple master mind-like game, based on a function designed by a student. The player clicks on two colors and the program responds with an answer that indicates how many colors and places were correct.

```
(master check-guess) → symbol?
  check-guess : (-> symbol? symbol? symbol? symbol? boolean?)
```

Chooses two “secret” colors and then opens a graphical user interface for playing *MasterMind*. The player is prompted to choose two colors, via a choice tablet and mouse clicks. Once chosen, `master` uses *`check-guess`* to compare them.

If the two guesses completely match the two secret colors, *`check-guess`* must return `'PerfectGuess`; otherwise it must return a different, informative symbol.

1.6 Simple Drawing: `"draw.ss"`

The teachpack provides two sets of functions: one for drawing into a canvas and one for reacting to canvas events.

Warning: *This teachpack is deprecated. Unless you’re solving exercises taken from How To Design Programs, we strongly encourage you to use the world teachpack instead; see §1.2 “Simulations and Animations: `"world.ss"`”.*

1.6.1 Drawing on a Canvas

DrawColor: `(and/c symbol? (one-of/c 'white 'yellow 'red 'blue 'green 'black))` These six colors are definitely provided. If you want other colors, guess! For example, `'orange` works, but `'mauve` doesn’t. If you apply the function to a symbol that it doesn’t recognize as a color, it raises an error.

```
(start width height) → true
  width : number?
  height : number?
```

Opens a *`width`* x *`height`* canvas.

```
(start/cartesian-plane width height) → true
  width : number?
  height : number?
```

Opens a `width` x `height` canvas and draws a Cartesian plane.

`(stop) → true`

Closes the canvas.

`(draw-circle p r c) → true`
  `p : posn?`
  `r : number?`
  `c` : DrawColor

Draws a `c` circle at `p` with radius `r`.

`(draw-solid-disk p r c) → true`
  `p : posn?`
  `r : number?`
  `c` : DrawColor

Draws a `c` disk at `p` with radius `r`.

`(draw-solid-rect ul width height c) → true`
  `ul : posn?`
  `width : number?`
  `height : number?`
  `c` : DrawColor

Draws a `width` x `height`, `c` rectangle with the upper-left corner at `ul`.

`(draw-solid-line strt end c) → true`
  `strt : posn?`
  `end : posn?`
  `c` : DrawColor

Draws a `c` line from `strt` to `end`.

`(draw-solid-string p s) → true`
  `p : posn?`
  `s : string?`

Draws `s` at `p`.

`(sleep-for-a-while s) → true`

`s : number?`

Suspends evaluation for `s` seconds.

The teachpack also provides `clear-` operations for each `draw-` operation. The arguments are the same. Note: use `clear-rectangle` instead of `clear-string` for now. The color argument for all `clear-` functions are optional.

1.6.2 Interactions with Canvas

`(wait-for-mouse-click) → posn?`

Waits for the user to click on the mouse, within the canvas.

DrawKeyEvent: `(or/c char? symbol?)` A DrawKeyEvent represents keyboard events:

- `char?`, if the user pressed an alphanumeric key;
- `symbol?`, if the user pressed, for example, an arror key: `'up 'down 'left 'right`

`(get-key-event) → (or/c false` DrawKeyEvent`)`

Checks whether the user has pressed a key within the window; `false` if not.

DrawWorld: For proper interactions, using the teachpack requires that you provide a data definition for DrawWorld . In principle, there are no constraints on this data definition. You can even keep it implicit, even if this violates the Design Recipe.

The following functions allow programs to react to events from the canvas.

`(big-bang n w) → true`
`n : number?`
`w :` DrawWorld

Starts the clock, one tick every `n` (fractal) seconds; `w` becomes the first “current” world.

`(on-key-event change) → true`
`change : (->` DrawKeyEvent DrawWorld DrawWorld`)`

Adds `change` to the world. The function reacts to keyboard events and creates a new DrawWorld.

```
(on-tick-event tock) → true
  tock : (-> DrawWorld DrawWorld)
```

Adds `tock` to the world. The function reacts to clock tick events, creating a new current world.

```
(end-of-time) → DrawWorld
```

Stops the world; returns the current world.

1.7 Hangman: `"hangman.ss"`

The teachpack implements the callback functions for playing a *Hangman* game, based on a function designed by a student. The player guesses a letter and the program responds with an answer that indicates how many times, if at all, the letter occurs in the secret word.

The teachpack provides all the drawing operations from §1.6 "Simple Drawing: `"draw.ss"`" for managing a canvas into which the "hangman" is drawn.

```
(hangman make-word reveal draw-next-part) → true
  make-word : (-> symbol? symbol? symbol? word?)
  reveal : (-> word? word? word?)
  draw-next-part : (-> symbol? true)
```

Chooses a "secret" three-letter word and uses the given functions to manage the *Hangman* game.

```
(hangman-list reveal-for-list
              draw-next-part) → true
  reveal-for-list : (-> symbol? (list-of symbol?) (list-of symbol?)
                        (list-of symbol?))
  draw-next-part : (-> symbol? true)
```

Chooses a "secret" word—a list of symbolic letters—and uses the given functions to manage the *Hangman* game: `reveal-for-list` determines how many times the chosen letter occurs in the secret word; `draw-next-part` is given the symbolic name of a body part and draws it on a separately managed canvas.

1.8 Managing Control Arrows: `"arrow.ss"`

The teachpack implements a controller for moving shapes across a canvass. A shape is a class of data for which `move` and `draw` operations can be drawn.

```
(control-left-right shape n move draw) → true
  shape : Shape
  n : number?
  move : (-> number? Shape Shape)
  draw : (-> Shape true)
```

Moves shape *n* pixels left (negative) or right (positive).

```
(control-up-down shape n move draw) → true
  shape : Shape
  n : number?
  move : (-> number? Shape Shape)
  draw : (-> Shape true)
```

Moves shape *n* pixels up (negative) or down (positive).

```
(control shape n move-lr move-ud draw) → true
  shape : Shape
  n : number?
  move-lr : (-> number? Shape Shape)
  move-ud : (-> number? Shape Shape)
  draw : (-> Shape true)
```

Moves shape `N` pixels left or right and up or down, respectively.

Example:

```
; A shape is a structure:
; (make-posn num num)

; RAD : the radius of the simple disk moving across a canvas
(define RAD 10)

; move : number shape -> shape or false
; to move a shape by delta according to translate
; effect: to redraw it
(define (move delta sh)
  (cond
    [(and (clear-solid-disk sh RAD)
```

```
            (draw-solid-disk (translate sh delta) RAD))
       (translate sh delta)]
      [else false]))

; translate : shape number -> shape
; to translate a shape by delta in the x direction
(define (translate sh delta)
  (make-posn (+ (posn-x sh) delta) (posn-y sh)))

; draw-it : shape -> true
; to draw a shape on the canvas: a disk with radius
(define (draw-it sh)
  (draw-solid-disk sh RAD))

; RUN:

; this creates the canvas
(start 100 50)

; this creates the controller GUI
(control-left-right (make-posn 10 20) 10 move draw-it)
```

1.9 Manipulating Simple HTML Documents: `"docs.ss"`

The teachpack provides three operations for creating simple “HTML” documents:

Annotation An Annotation is a symbol that starts with “<” and ends in “>”. An end annotation is one that starts with “</”.

`(atom? x)` → `boolean?`
  `x : any/c`

Determines whether or not a Scheme value is a number, a symbol, or a string.

`(annotation? x)` → `boolean?`
  `x : any/c`

Determines whether or not a Scheme symbol is a document annotation.

`(end-annotation x)` → Annotation
  `x` : Annotation

Consumes an annotation and produces a matching ending annotation.

```
(write-file l) → true
  l : (list-of atom)
```

Consumes a list of symbols and annotations and prints them out as a ”file”.

Sample session: set teachpack to “docs.ss”> and click RUN:

```
  > (annotation? 0)
  false
  > (annotation? '<bold>)
  true
  > (end-annotation 0)
  end-annotation: not an annotation: 0
  > (write-file (list 'a 'b))
  a b
```

1.10 Working with Files and Directories: `"dir.ss"`

The teachpack provides structures and operations for working with files and directories:

```
(struct dir (name dirs files))
  name : string?
  dirs : (list-of dir?)
  files : (list-of file?)
```

```
(struct file (name content))
  name : string?
  content : (list-of char?)
```

```
(create-dir path) → dir?
  path : string?
```

Turns the directory found at *path* on your computer into an instance of `dir?`.

Sample: Set teachpack to <code>dir.ss</code> and click RUN:

```
  > (create-dir ".")
  (make-dir
    '|.|
    empty
    (cons (make-file 'ball1.gif 1289 empty)
          (cons (make-file 'blueball.gif 205 empty)
```

```
(cons (make-file 'greenbal.gif 204 empty)
      (cons (make-file 'redball.gif 203 empty)
            (cons (make-file 'ufo.gif 1044 empty)
                  (cons (make-file 'gif-test.ss 5811 empty)
                        empty)))))))
```

Using "." usually means the directory in which your program is located. In this case, the directory contains no sub-directories and six files.

Note: Softlinks are always treated as if they were empty files.

1.11 Graphing Functions: `"graphing.ss"`

The teachpack provides two operations for graphing functions in the regular (upper right) quadrant of the Cartesian plane (between 0 and 10 in both directions):

```
(graph-fun f color) → true
  f : (-> number? number?)
  color : symbol?
```

Draws the graph of `f` with the given `color`.

```
(graph-line line color) → true
  line : (-> number? number?)
  color : symbol?
```

Draws `line`, a function representing a straight line, with a given color.

For color symbols, see §1.6 "Simple Drawing: `"draw.ss"`".

1.12 Simple Graphical User Interfaces: `"gui.ss"`

The teachpack provides operations for creating and manipulating graphical user interfaces. We recommend using the world teachpack instead.

Window A Window is a data representation of a visible window on your computer screen.

GUI-ITEM A GUI-Item is a data representation of an active component of a window on your computer screen.

```
(create-window g) → Window
  g : (listof (listof GUI-ITEM))
```

Creates a window from the “matrix” of gui items *g*.

```
(window? x) → boolean?
  x : any/c
```

Is the given value a window?

```
(show-window w) → true
  w : Window
```

Shows *w*.

```
(hide-window w) → true
  w : window
```

Hides *w*.

```
(make-button label callback) → GUI-ITEM
  label : string>
  callback : (-> event%  boolean)
```

Creates a button with *label* and *callback* function. The latter receives an argument that it may safely ignore.

```
(make-message msg) → GUI-ITEM
  msg : string?
```

Creates a message item from *msg*.

```
(draw-message g m) → true
  g : GUI-ITEM
  m : string?
```

Displays *m* in message item *g* and erases the current message.

```
(make-text txt) → GUI-ITEM
  txt : string?
```

Creates an text editor (with label *txt*) that allows users to enter text.

```
(text-contents g) → string?
  g : GUI-ITEM
```

Determines the current contents of a text GUI-ITEM.

`(make-choice choices)` → GUI-ITEM
  `choices : (listof string?)`

Creates a choice menu from `choices` that permits users to choose from some alternatives.

`(choice-index g)` → `natural-number/c`
  `g` : GUI-ITEM

Determines the choice that is currently selected in a choice GUI-ITEM; the result is the 0-based index in the choice menu

Example 1:

```
> (define w
    (create-window
      (list (list (make-button "QUIT" (lambda (e) (hide-window w)))))))
; A button appears on the screen.
; Click on the button and it will disappear.
> (show-window w)
; The window disappears.
```

Example 2:

```
; text1 : GUI-ITEM
(define text1
  (make-text "Please enter your name"))

; msg1 : GUI-ITEM
(define msg1
  (make-message (string-append "Hello, World" (make-string 33 #\space))))

; Event -> true
; draws the current contents of text1 into msg1, prepended with "Hello, "
(define (respond e)
  (draw-message msg1 (string-append "Hello, " (text-contents text1))))

; set up window with three "lines":
; a text field, a message, and two buttons
; fill in text and click OKAY
(define w
 (create-window
  (list
   (list text1)
   (list msg1)
```

```
    (list (make-button "OKAY" respond)
          (make-button "QUIT" (lambda (e) (hide-window w))))))
```

1.13 An Arrow GUI: `"arrow-gui.ss"`

The teachpack provides operations for creating and manipulating an arrow GUI. We recommend using the world teachpack instead.

modelT `(-> button% event% true)`

A modelT is a function that accepts and ignores two arguments.

`(control) → symbol?`

Reads out the current state of the message field.

`(view s) → true`
  `s : (or/c string? symbol?)`

Displays `s` in the message field.

`(connect l r u d) → true`
  `l` : modelT
  `r` : modelT
  `u` : modelT
  `d` : modelT

Connects four controllers with the four directions in the arrow window.

Example:

```
; Advanced
(define (make-model dir)
   (lambda (b e)
     (begin
       (view dir)
       (printf "~a ~n" (control)))))

(connect
   (make-model "left")
   (make-model "right")
   (make-model "up")
   (make-model "down"))
```

Now click on the four arrows. The message field contains the current direction, the print-out the prior contents of the message field.

1.14 Controlling an Elevator: `"elevator.ss"`

The teachpack implements an elevator simulator.

It displays an eight-floor elevator and accepts mouse clicks from the user, which are translated into service demands for the elevator.

```
(run NextFloor) → any/c
  NextFloor : number?
```

Creates an elevator simulator that is controlled by *`NextFloor`*. This function consumes the current floor, the direction in which the elevator is moving, and the current demands. From that, it computes where to send the elevator next.

Example: Define a function that consumes the current state of the elevator (three arguments) and returns a number between 1 and 8. Here is a non-sensical definition:

```
(define (controller x y z) 7)
```

It moves the elevator once, to the 7th floor.

Second, set the teachpack to <code>elevator.ss</code>, click RUN, and evaluate

```
(run controller)
```

1.15 Queens: `"show-queen.ss"`

The teachpack provides the operation `show-queen`, which implements a GUI for exploring the n-queens problem.

```
(show-queen board) → true
  board : (list-of (list-of boolean?))
```

The function `show-queen` consumes a list of lists of booleans that describes a *`board`*. Each of the inner lists must have the same length as the outer list. The `true`s correspond to positions where queens are, and the `false`s correspond to empty squares. The function returns nothing.

In the GUI window that `show-queen` opens, the red and orange dots show where the queens are. The green dot shows where the mouse cursor is. Each queen that threatens the green spot is shown in red, and the queens that do not threaten the green spot are shown in orange.

1.16 Matrix Operations: `"matrix.ss"`

The experimental teachpack supports matrices and matrix operations. A matrix is just a rectangle of 'objects'. It is displayed as an image, just like the images from §1.1 "Manipulating Images: `"image.ss"`". Matrices are images and, indeed, scenes in the sense of the §1.2 "Simulations and Animations: `"world.ss"`".

No educational materials involving matrices exist.

The operations access a matrix in the usual (school-mathematics) manner: row first, column second.

The operations aren't tuned for efficiency so don't expect to build programs that process lots of data.

Rectangle A Rectangle (of X) is a non-empty list of lists containing X where all elements of the list are lists of equal (non-zero) length.

```
(matrix? o) → boolean?
  o : any/c
```

determines whether the given object is a matrix?

```
(matrix-rows m) → natural-number/c
  m : matrix?
```

determines how many rows this matrix `m` has

```
(matrix-cols m) → natural-number/c
  m : matrix?
```

determines ow many columns this matrix `m` has

```
(rectangle->matrix r) → matrix?
  r : Rectangle
```

creates a matrix from the given Rectangle

```
(matrix->rectangle m) → Rectangle
  m : matrix?
```

creates a rectangle from this matrix `m`

```
(make-matrix n m l) → matrix?
  n : natural-number/c
  m : natural-number/c
  l : (Listof X)
```

creates an `n` by `m` matrix from `l`

NOTE: `make-matrix` would consume an optional number of entries, if it were like `make-vector`

```
(build-matrix n m f) → matrix?
  n : natural-number/c
  m : natural-number/c
  f : (-> (and/c natural-number/c (</c m))
          (and/c natural-number/c (</c n))
          any/c)
```

creates an `n` by `m` matrix by applying `f` to `(0 ,0)`, `(0 ,1)`, ..., `((sub1 m) ,(sub1 n))`

```
(matrix-ref m i j) → any/c
  m : matrix?
  i : (and/c natural-number/c (</c (matrix-rows m)))
  j : (and/c natural-number/c (</c (matrix-rows m)))
```

retrieve the item at `(i,j)` in matrix `m`

```
(matrix-set m i j x) → matrix?
  m : matrix?
  i : (and/c natural-number/c (</c (matrix-rows m)))
  j : (and/c natural-number/c (</c (matrix-rows m)))
  x : any/c
```

creates a new matrix with `x` at `(i,j)` and all other places the same as in `m`

```
(matrix-where? m pred?) → (listof posn?)
  m : matrix?
  pred? : (-> any/c boolean?)
```

`(matrix-where? M P)` produces a list of `(make-posn i j)` such that `(P (matrix-ref M i j))` holds

```
(matrix-render m) → Rectangle
  m : matrix?
```

renders this matrix `m` as a rectangle of strings

```
(matrix-minor m i j) → matrix?
  m : matrix?
  i : (and/c natural-number/c (</c (matrix-rows m)))
  j : (and/c natural-number/c (</c (matrix-rows m)))
```

creates a matrix minor from `m` at (`i`,`j`)

```
(matrix-set! m i j x) → matrix?
  m : matrix?
  i : (and/c natural-number/c (</c (matrix-rows m)))
  j : (and/c natural-number/c (</c (matrix-rows m)))
  x : any/c
```

like `matrix-set` but uses a destructive update

2 HtDC Teachpacks

2.1 Geometry: geometry.*

Add

```
import geometry.*
```

at the top of your Definitions Window to import this library.

This package provides a class for representing positions in a Cartesian world:

```
+------------+
| Posn       |
+------------+
| int x      |
| int y      |
+------------+
```

Posn is a class with two fields, one per coordinate. The constructor consumes two integers.

2.2 Colors: colors.*

Add

```
import colors.*
```

at the top of your Definitions Window to import this library.

This package provides classes for representing colors:

```
                      +---------+
                      | IColor  |
                      +---------+
                           |
                          / \
                          ---
                           |
    -------------------------------------------------------
    |           |          |          |          |        |
+--------+  +--------+ +--------+ +--------+ +--------+  +--------+
| Blue   |  | Green  | | Red    | | White  | | Yellow|  | Black  |
+--------+  +--------+ +--------+ +--------+ +--------+  +--------+
```

IColor is an interface. Its variants are created with no arguments.

2.3 Draw: draw.*

Add

```
import draw.*
```

at the top of your Definitions Window to import this library.

This package provides classes and methods for a visual world. Here is its class diagram of public fields and methods:

```
import colors.*;
import geometry.*;

 +-----------------------------------+
 | abstract World                    |
 +-----------------------------------+
 | Canvas theCanvas                  |----+
 +-----------------------------------+    |
 | boolean bigBang(int,int,double)   |    |
 | boolean endOfTime(String)         |    |
 | World endOfWorld(String)          |    |
 |                                   |    |
 |                                   |    |
 | abstract World onTick()           |    |
 | abstract World onKeyEvent(String) |    |
 | abstract boolean draw()           |    |
 +-----------------------------------+    |
                                          |
                                          v
    +---------------------------------------+
    | Canvas                                |
    +---------------------------------------+
    +---------------------------------------+
    | boolean show()                        |
    | boolean close()                       |
    | boolean drawCircle(Posn,int,IColor)   |
    | boolean drawDisk(Posn,int,IColor)     |
    | boolean drawRect(Posn,int,int,IColor) |
    | boolean drawLine(Posn,Posn,IColor)    |
    | boolean drawString(Posn,String)       |
    +---------------------------------------+
```

Methods in these classes may fail due to the unavailability of the physical devices, inappropriate uses, etc. In those cases, they fail with an exception.

2.3.1 World

The abstract `World` class exports the following methods.

`bigBang : (int width,int height,double speed)`

Initializes the world, associates it with a `width` x `height` `Canvas`, displays this canvas, enables keyevents, and finally starts the clock at a rate of one tick per `speed` seconds. If it succeeds with all of its actions, the method produces `true`.

Note: `width`, `height` and `speed` must be a positive.

The canvas in `World` is called

`theCanvas`.

References to a ”canvas” in conjunction with the `World` class denote this default canvas.

`endOfTime : ()`

Displays the given message, stops the clock and, if it succeeds, produces `true`. After the end of time, events no longer trigger calls to `onTick` or `onKeyEvent`. The canvas remains visible.

`endOfWorld : (String msg)`

Displays the given message, stops the clock and, if it succeeds, produces the last `World`. After the end of the world, events no longer trigger calls to `onTick` or `onKeyEvent` (see below). The canvas remains visible.

A derived concrete class must supply definitions for the following methods:

`onTick : ()`

Invoked for every tick of the clock. Its purpose is to create a `World` whose differences with `this` one represent what happened during the amount of time it takes the clock to tick.

`onKeyEvent : (String key)`

Invoked for every keyboard event associated with the canvas. Its purposes is to create a `World` whose differences with `this` one represent what happens due to the user's use of the keyboard. The latter is represented with the string-valued argument `key`.

`draw : ()`

Invoked *after* one of the two event handlers has been called. Its purpose is to present `this` `World` graphically on its canvas. If it succeeds, its result is `true`.

A program may, in principle, start several instances of (subclasses of) `World`. If it does, the event handlers are called in a unpredictable order.

2.3.2 Canvas

To create an instance of the `Canvas` class, a program must supply two `int` values: one for the width of the canvas and one for its height. The canvas is a rectangle, whose borders are parallel to the computer screen's borders. A program can use the following methods on instances of `Canvas`]

`show : ()`

Initializes the canvas to a white area, enables the drawing methods, and finally displays the canvas. If it succeeds, it produces `true`. Invoking the method a second time without calling `close` before has no effect.

`close : ()`

Hides the canvas and erases the current content. If it succeeds, it produces `true`.

Closing the Canvas using the display controls does not fully hide the canvas; it is still necessary to invoke `close` before `show` is re-enabled.

`drawCircle : (Posn p,int r,IColor c)`

Draws a circle on `this`Canvas] at `p` with radius `r` and color `c`. If it succeeds, it produces `true`.

`drawDisk : (Posn p,int r,IColor c)`

Draws a disk on `this`Canvas] at `p` with radius `r` and color `c`. If it succeeds, it produces `true`.

`drawRect : (Posn p,int w,int h,IColor c)`

Draws a solid rectangle on `this`Canvas] at `p` with width `w`, height `h`, and color `c`. The rectangle's lines are parallel to the canvas's borders. If it succeeds, it produces `true`.

`drawLine : (Posn p0,Posn p1,IColor c)`

Draws a line on `this`Canvas] from `p0` to `p1` using color `c`. If it succeeds, it produces `true`.

`drawString : (Posn p,String s)`

Draws the string `s` at `p` on `this`Canvas]. If it succeeds, it produces `true`.

2.4 Draw: idraw.*

Add

```
import idraw.*
```

at the top of your Definitions Window to import this library.

This package provides stateful classes and imperative methods for a visual world. Here is its class diagram of public fields and methods:

```
import colors.*;
import geometry.*;

  +-----------------------------------+
  | abstract World                    |
  +-----------------------------------+
  | Canvas theCanvas                  |---+
  +-----------------------------------+   |
  | void bigBang(int,int,double)      |   |
  | World endOfTime(String)           |   |
  | World endOfWorld(String)          |   |
  | abstract void onTick()            |   |
  | abstract void onKeyEvent(String)  |   |
  | abstract void draw()              |   |
  +-----------------------------------+   |
                                          |
                                          v
          +-------------------------------------+
          | Canvas                              |
```

```
+------------------------------------+
+------------------------------------+
| void show()                        |
| void close()                       |
| void drawCircle(Posn,int,IColor)   |
| void drawDisk(Posn,int,IColor)     |
| void drawRect(Posn,int,int,IColor) |
| void drawLine(Posn,Posn,IColor)    |
| void drawString(Posn,String)       |
+------------------------------------+
```

The abstract `World` class in `idraw` provides the same methods as the `World` class in §2.3.1 “World” (`draw` package). Their return values are usually `void`, however, except for `endOfTime` and `endOfWorld`, which continue to return the last world.

In an analogous manner, the methods in the `Canvas` class export the same methods as the `Canvas` class in §2.3.2 “Canvas” (`draw` package). Again their return values are `void`.

3 HtDP/2e Teachpacks

3.1 Worlds and the Universe

```
(require 2htdp/universe)
```

This `universe.ss` teachpack implements and provides the functionality for creating interactive, graphical programs that consist of plain mathematical functions. We refer to such programs as *world* programs. In addition, world programs can also become a part of a *universe*, a collection of worlds that can exchange messages.

The purpose of this documentation is to give experienced Schemers and HtDP teachers a concise overview for using the library. The first part of the documentation focuses on world programs. Section §3.1.5 “A First Sample World” presents an illustration of how to design such programs for a simple domain; it is suited for a novice who knows how to design conditional functions for symbols. The second half of the documentation focuses on ”universe” programs: how it is managed via a server, how world programs register with the server, etc. The last two sections show how to design a simple universe of two communicating worlds.

Note: For a quick and educational introduction to just worlds, see How to Design Programs, Second Edition: Prologue. As of August 2008, we also have a series of projects available as a small booklet on How to Design Worlds.

3.1.1 Basics

The teachpack assumes working knowledge of the basic image manipulation primitives and supports several functions that require a special kind of image, called a *scene*, which is an image whose pinholes are at position (0, 0). For example, the teachpack displays only scenes in its canvas.

```
(scene? x) → boolean?
  x : any/c
```

determines whether *x* is a scene.

```
(empty-scene width height) → scene?
  width : natural-number/c
  height : natural-number/c
```

creates a plain white, *width* x *height* scene.

```
(place-image img x y s) → scene?
```

```
img : image?
x : number?
y : number?
s : scene?
```

creates a scene by placing `img` at (`x`, `y`) into `s`; (`x`, `y`) are computer graphics coordinates, i.e., they count right and down from the upper-left corner.

3.1.2 Simple Simulations

The simplest kind of animated world program is a time-based simulation, which is a series of scenes. The programmer's task is to supply a function that creates a scene for each natural number. By handing this function to the teachpack displays the simulation.

```
(run-simulation create-image) → true
  create-image : (-> natural-number/c scene)
```

opens a canvas and starts a clock that tick 28 times per second. Every time the clock ticks, DrScheme applies `create-image` to the number of ticks passed since this function call. The results of these function calls are displayed in the canvas. The simulation runs until you click the `Stop` button in DrScheme or close the window. At that point, `run-simulation` returns the number of ticks that have passed.

Example:

```
(define (create-UFO-scene height)
  (place-image UFO 50 height (empty-scene 100 100)))

(define UFO
  (overlay (circle 10 'solid 'green)
           (rectangle 40 4 'solid 'green)))

(run-simulation create-UFO-scene)
```

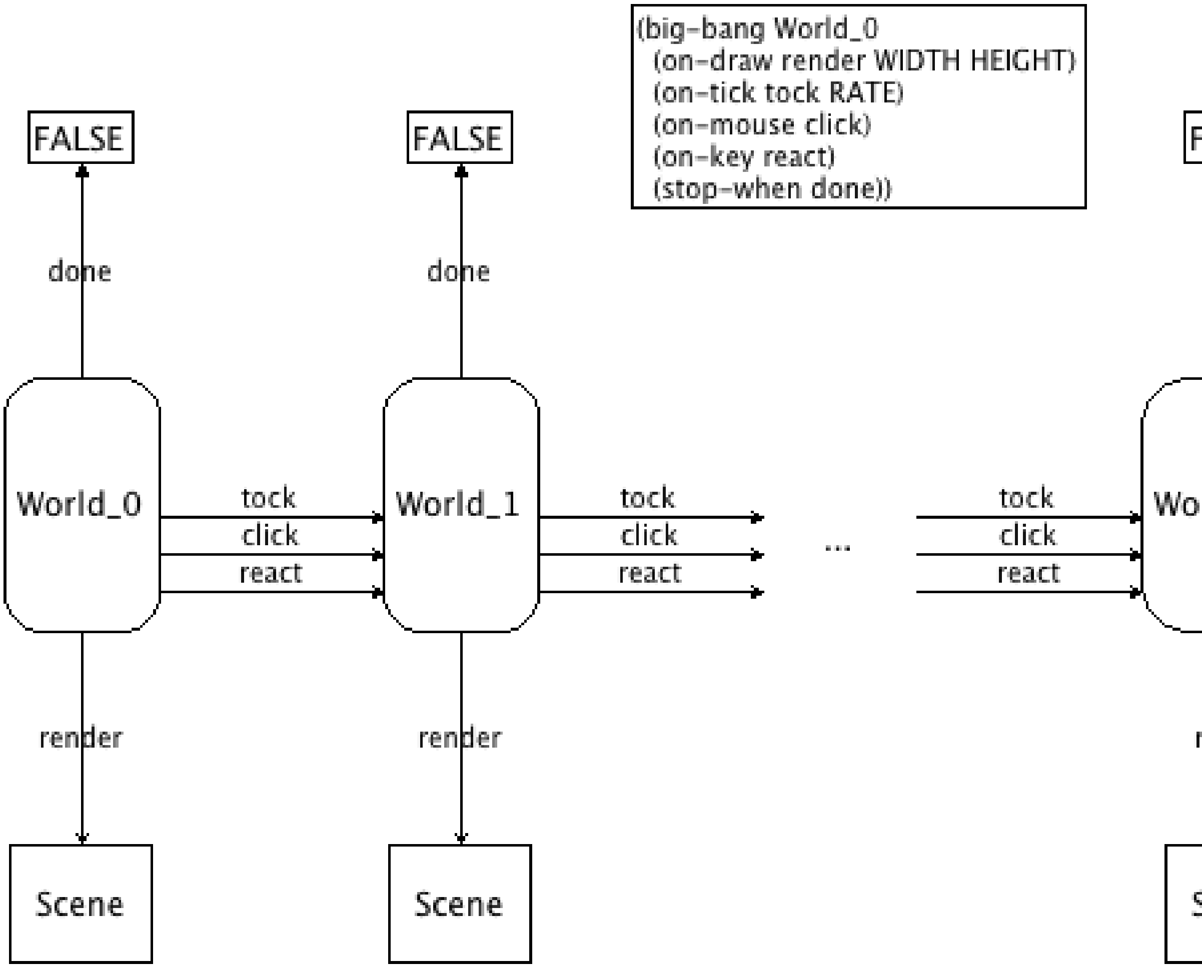
3.1.3 Interactions

The step from simulations to interactive programs is relatively small. Roughly speaking, a simulation designates one function, `create-image`, as a handler for one kind of event: clock ticks. In addition to clock ticks, world programs can also deal with two other kinds of events: keyboard events and mouse events. A keyboard event is triggered when a computer user presses or releases a key on the keyboard. Similarly, a mouse event is the movement of the mouse, a click on a mouse button, the crossing of a boundary by a mouse movement, etc.

Your program may deal with such events via the *designation* of *handler* functions. Specifically, the teachpack provides for the installation of three event handlers: `on-tick`, `on-key`, and `on-mouse`. In addition, a world program may specify a `draw` function, which is called every time your program should visualize the current world, and a `done` predicate, which is used to determine when the world program should shut down.

Each handler function consumes the current state of the world and optionally a data representation of the event. It produces a new state of the world.

The following picture provides an intuitive overview of the workings of a world program in the form of a state transition diagram.



The `big-bang` form installs `World_0` as the initial WorldState. The handlers `tock`, `react`, and `click` transform one world into another one; each time an event is handled, `done` is used to check whether the world is final, in which case the program is shut down; and finally, `draw` renders each world as a scene, which is then displayed on an external canvas.

WorldState : `any/c`

The design of a world program demands that you come up with a data definition of all possible states. We use WorldState to refer to this collection of data, using a capital W to distinguish it from the program. In principle, there are no constraints on this data definition though it mustn't be an instance of the Package structure (see below). You can even keep it implicit, even if this violates the Design Recipe.

```
(big-bang state-expr clause ...)

clause = (on-tick tick-expr)
       | (on-tick tick-expr rate-expr)
       | (on-key key-expr)
       | (on-mouse key-expr)
       | (on-draw draw-expr)
       | (on-draw draw-expr width-expr height-expr)
       | (stop-when stop-expr)
       | (record? boolean-expr)
       | (on-receive rec-expr)
       | (register IP-expr)
       | (name name-expr)
```

starts a world program in the initial state specified with `state-expr`, which must of course evaluate to an element of WorldState. Its behavior is specified via the handler functions designated in the optional `spec` clauses, especially how the world program deals with clock ticks, with key events, with mouse events, and eventually with messages from the universe; how it renders itself as a scene; when the program must shut down; where to register the world with a universe; and whether to record the stream of events. A world specification may not contain more than one `on-tick`, `on-draw`, or `register` clause. A `big-bang` expression returns the last world when the stop condition is satisfied (see below) or when the programmer clicks on the `Stop` button or closes the canvas.

- `(on-tick tick-expr)`

  `tick-expr` : `(->` WorldState WorldState`)`

  tell DrScheme to call the `tick-expr` function on the current world every time the clock ticks. The result of the call becomes the current world. The clock ticks at the rate of 28 times per second.

- ---

  `(on-tick tick-expr rate-expr)`

  `tick-expr : (-> WorldState WorldState)`
  `rate-expr : natural-number/c`

  tell DrScheme to call the `tick-expr` function on the current world every time the clock ticks. The result of the call becomes the current world. The clock ticks at the rate of `rate-expr`.

- A KeyEvent represents key board events, e.g., keys pressed or released.

  KeyEvent : `(or/c char? symbol?)`

  A character is used to signal that the user has hit an alphanumeric key. A symbol denotes arrow keys or special events:

  - `'left` is the left arrow,
  - `'right` is the right arrow,
  - `'up` is the up arrow,
  - `'down` is the down arrow, and
  - `'release` is the event of releasing a key.

`(key-event? x) → boolean?`
`x : any`

determines whether `x` is a KeyEvent

`(key=? x y) → boolean?`
`x : key-event?`
`y : key-event?`

compares two KeyEvent for equality

`(on-key change-expr)`

`change-expr : (-> WorldState key-event? WorldState)`

tell DrScheme to call `change-expr` function on the current world and a KeyEvent for every keystroke the user of the computer makes. The result of the call becomes the current world.

Here is a typical key-event handler:

```
(define (change w a-key)
  (cond
    [(key=? a-key 'left)  (world-go w -DELTA)]
```

```
  [(key=? a-key 'right) (world-go w +DELTA)]
  [(char? a-key) w] ; to demonstrate order-free checking
  [(key=? a-key 'up)    (world-go w -DELTA)]
  [(key=? a-key 'down)  (world-go w +DELTA)]
  [else w]))
```

The omitted, auxiliary function *world-go* is supposed to consume a world and a number and produces a world.

- A MouseEvent represents mouse events, e.g., mouse movements or mouse clicks, by the computer's user.

  MouseEvent : `(one-of/c 'button-down 'button-up 'drag 'move 'enter 'leave)`

  All MouseEvents are represented via symbols:

  - `'button-down` signals that the computer user has pushed a mouse button down;
  - `'button-up` signals that the computer user has let go of a mouse button;
  - `'drag` signals that the computer user is dragging the mouse;
  - `'move` signals that the computer user has moved the mouse;
  - `'enter` signals that the computer user has moved the mouse into the canvas area; and
  - `'leave` signals that the computer user has moved the mouse out of the canvas area.

`(mouse-event? x)` → `boolean?`

`x : any`

determines whether `x` is a KeyEvent

`(mouse=? x y)` → `boolean?`

`x : mouse-event?`

`y : mouse-event?`

compares two KeyEvent for equality

`(on-mouse clack-expr)`

`clack-expr` : `(->` WorldState `natural-number/c natural-number/c` MouseEvent WorldState`)`

tell DrScheme to call `clack-expr` on the current world, the current `x` and `y` coordinates of the mouse, and and a MouseEvent for every (noticeable) action of the mouse by the computer user. The result of the call becomes the current world.

Note: the computer's software doesn't really notice every single movement of the mouse (across the mouse pad). Instead it samples the movements and signals most of them.

- ---

  `(on-draw render-expr)`

  `render-expr : (-> WorldState scene?)`

  tell DrScheme to call the function *`render-expr`* whenever the canvas must be drawn. The external canvas is usually re-drawn after DrScheme has dealt with an event. Its size is determined by the size of the first generated scene.

`(on-draw render-expr width-expr height-expr)`

```
render-expr : (-> WorldState scene?)
width-expr : natural-number/c
height-expr : natural-number/c
```

tell DrScheme to use a *`width-expr`* by *`height-expr`* canvas instead of one determine by the first generated scene.

- ---

  `(stop-when last-world?)`

  `last-world? : (-> WorldState boolean?)`

  tell DrScheme to call the *`last-world?`* function whenever the canvas is drawn. If this call produces `true`, the world program is shut down. Specifically, the clock is stopped; no more tick events, KeyEvents, or MouseEvents are forwarded to the respective handlers. The `big-bang` expression returns this last world.

- ---

  `(record? boolean-expr)`

  `boolean-expr : boolean?`

  tell DrScheme to record all events and to enable a replay of the entire interaction. The replay action also generates one png image per scene and an animated gif for the entire sequence.

The following example shows that `(run-simulation create-UFO-scene)` is a shorthand for three lines of code:

```
(define (create-UFO-scene height)
  (place-image UFO 50 height (empty-scene 100 100)))

(define UFO
  (overlay (circle 10 'solid 'green)
           (rectangle 40 4 'solid 'green)))
```

```
(big-bang 0
          (on-tick add1)
          (on-draw create-UFO-scene))
```

Exercise: Add a condition for stopping the flight of the UFO when it reaches the bottom.

3.1.4 Scenes and Images

For the creation of scenes from the world, use the functions from §1.1 "Manipulating Images: `"image.ss"`". The teachpack adds the following two functions, which are highly useful for creating scenes.

```
(nw:rectangle width height solid-or-filled c) → image?
  width : natural-number/c
  height : natural-number/c
  solid-or-filled : Mode
  c : Color
```

creates a `width` by `height` rectangle, solid or outlined as specified by `solid-or-filled` and colored according to `c`, with a pinhole at the upper left corner.

```
(scene+line s x0 y0 x1 y1 c) → scene?
  s : scene?
  x0 : number?
  y0 : number?
  x1 : number?
  y1 : number?
  c : Color
```

creates a scene by placing a line of color `c` from (`x0`, `y0`) to (`x1`, `y1`) using computer graphics coordinates. In contrast to the `add-line` function, `scene+line` cuts off those portions of the line that go beyond the boundaries of the given `s`.

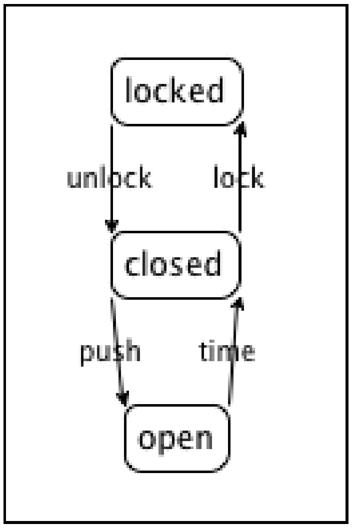
3.1.5 A First Sample World

This section uses a simple example to explain the design of worlds. The first subsection introduces the sample domain, a door that closes automatically. The second subsection is about the design of world programs in general, the remaining subsections implement a simulation of the door.

Understanding a Door

Say we wish to design a world program that simulates the working of a door with an automatic door closer. If this kind of door is locked, you can unlock it with a key. While this doesn't open the door per se, it is now possible to do so. That is, an unlocked door is closed and pushing at the door opens it. Once you have passed through the door and you let go, the automatic door closer takes over and closes the door again. When a door is closed, you can lock it again.

Here is a diagram that translates our words into a graphical representation:



Like the picture of the general workings of a world program, this diagram displays a so-called "state machine." The three circled words are the states that our informal description of the door identified: locked, closed (and unlocked), and open. The arrows specify how the door can go from one state into another. For example, when the door is open, the automatic door closer shuts the door as time passes. This transition is indicated by the arrow labeled "time passes." The other arrows represent transitions in a similar manner:

- "push" means a person pushes the door open (and let's go);
- "lock" refers to the act of inserting a key into the lock and turning it to the locked position; and
- "unlock" is the opposite of "lock."

Hints on Designing Worlds

Simulating any dynamic behavior via a world program demands two different activities. First, we must tease out those portions of our domain that change over time or in reaction to actions, and we must develop a data representation for this information. This is what we call WorldState. Keep in mind that a good data definition makes it easy for readers to map data to information in the real world and vice versa. For all others aspects of the world, we use global constants, including graphical or visual constants that are used in conjunction with the rendering operations.

Second, we must translate the actions in our domain—the arrows in the above diagram—into interactions with the computer that the universe teachpack can deal with. Once we have decided to use the passing of time for one aspect, key presses for another, and mouse movements for a third, we must develop functions that map the current state of the world—represented as data from WorldState—into the next state of the world. Put differently, we have just created a wish list with three handler functions that have the following general contract and purpose statements:

```
; tick : WorldState -> WorldState
; deal with the passing of time
(define (tick w) ...)

; click : WorldState Number Number MouseEvent -> WorldState
; deal with a mouse click at (x,y) of kind me
; in the current world w
(define (click w x y me) ...)

; control : WorldState KeyEvent -> WorldState
; deal with a key event (symbol, char) ke
; in the current world w
(define (control w ke) ...)
```

That is, the contracts of the various handler designations dictate what the contracts of our functions are, once we have defined how to represent the domain with data in our chosen language.

A typical program does not use all three of these functions. Furthermore, the design of these functions provides only the top-level, initial design goal. It often demands the design of many auxiliary functions. The collection of all these functions is your world program.

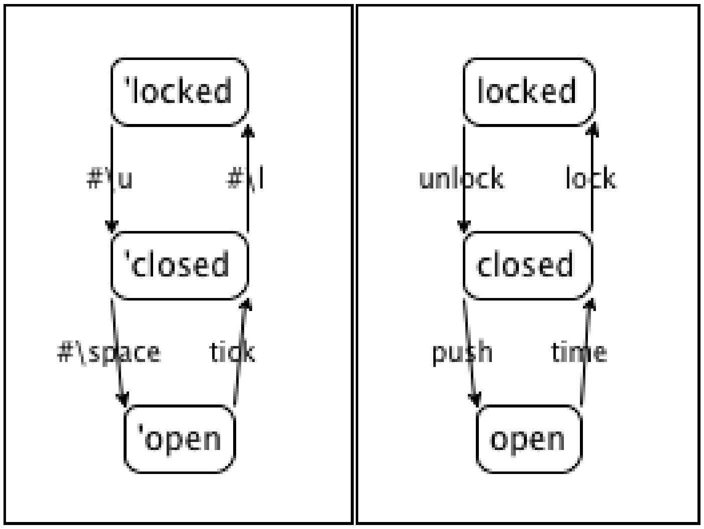
Simulating a Door: Data

Our first and immediate goal is to represent the world as data. In this specific example, the world consists of our door and what changes about the door is whether it is locked, unlocked but closed, or open. We use three symbols to represent the three states:

```
; WorldState is one of:
; -- ’locked
; -- ’closed
; -- ’open
; interpretation: state of door
```

Symbols are particularly well-suited here because they directly express the state of the door.

Now that we have a data definition, we must also decide which computer interactions should model the various actions on the door. Our pictorial representation of the door’s states and transitions, specifically the arrow from `open` to `closed` suggests the use of a function that simulates time. For the other three arrows, we could use either keyboard events or mouse clicks or both. Our solution uses three keystrokes: `#\u` for unlocking the door, `#\l` for locking it, and `#\space` for pushing it open. We can express these choices graphically by translating the above state-machine diagram from the world of information into the world of data.



For completeness, we have repeated the original diagram on the right so that you can see which computer interaction corresponds to which domain action.

Simulating a Door: Functions

Our analysis and data definition leaves us with three functions to design:

- *automatic-closer*, which closes the time during one tick;

- *door-actions*, which manipulates the time in response to pressing a key; and
- *render*, which translates the current state of the door into a visible scene.

Let's start with *automatic-closer*. Since *automatic-closer* acts as the `on-tick` handler, we get its contract, and it is easy to refine the purpose statement, too:

```
; automatic-closer : WorldState -> WorldState
; closes an open door over the period of one tick
(define (automatic-closer state-of-door) ...)
```

Making up examples is trivial when the world can only be in one of three states:

given state	desired state
`'locked`	`'locked`
`'closed`	`'closed`
`'open`	`'closed`

```
; automatic-closer : WorldState -> WorldState
; closes an open door over the period of one tick

(check-expect (automatic-closer 'locked) 'locked)
(check-expect (automatic-closer 'closed) 'closed)
(check-expect (automatic-closer 'open) 'closed)

(define (automatic-closer state-of-door) ...)
```

The template step demands a conditional with three clauses:

```
(define (automatic-closer state-of-door)
  (cond
    [(symbol=? 'locked state-of-door) ...]
    [(symbol=? 'closed state-of-door) ...]
    [(symbol=? 'open state-of-door) ...]))
```

The examples basically dictate what the outcomes of the three cases must be:

```
(define (automatic-closer state-of-door)
  (cond
    [(symbol=? 'locked state-of-door) 'locked]
    [(symbol=? 'closed state-of-door) 'closed]
    [(symbol=? 'open state-of-door) 'closed]))
```

Don't forget to run the example-tests.

For the remaining three arrows of the diagram, we design a function that reacts to the three chosen keyboard events. As mentioned, functions that deal with keyboard events consume both a world and a keyevent:

```
; door-actions : WorldState KeyEvent -> WorldState
; key events simulate actions on the door
(define (door-actions s k) ...)
```

given state	given keyevent	desired state
`'locked`	`#\u`	`'closed`
`'closed`	`#\l`	`'locked`
`'closed`	`#\space`	`'open`
`'open`	—	`'open`

The examples combine what the above picture shows and the choices we made about mapping actions to keyboard events.

From here, it is straightforward to turn this into a complete design:

```
(define (door-actions s k)
  (cond
    [(and (symbol=? 'locked s) (key=? #\u k)) 'closed]
    [(and (symbol=? 'closed s) (key=? #\l k)) 'locked]
    [(and (symbol=? 'closed s) (key=? #\space k)) 'open]
    [else s]))

(check-expect (door-actions 'locked #\u) 'closed)
(check-expect (door-actions 'closed #\l) 'locked)
(check-expect (door-actions 'closed #\space) 'open)
(check-expect (door-actions 'open 'any) 'open)
(check-expect (door-actions 'closed 'any) 'closed)
```

Last but not least we need a function that renders the current state of the world as a scene. For simplicity, let's just use a large text for this purpose:

```
; render : WorldState -> scene
; translate the current state of the door into a large text
(define (render s)
  (text (symbol->string s) 40 'red))

(check-expect (render 'closed) (text "closed" 40 'red))
```

The function `symbol->string` translates a symbol into a string, which is needed because `text` can deal only with the latter, not the former. A look into the language documentation revealed that this conversion function exists, and so we use it.

Once everything is properly designed, it is time to *run* the program. In the case of the universe teachpack, this means we must specify which function takes care of tick events, key events, and drawing:

```
(big-bang 'locked
          (on-tick automatic-closer)
          (on-key door-actions)
          (on-draw render))
```

Now it's time for you to collect the pieces and run them in DrScheme to see whether it all works.

Exercise: Design a data representation that closes the door over two (or three or more) clock ticks instead of one.

3.1.6 The World is not Enough

The library facilities covered so far are about designing individual programs with interactive graphical user interfaces (simulations, animations, games, etc.). In this section, we introduce capabilities for designing a distributed program, which is really a number of programs that coordinate their actions in some fashion. Each of the individual programs may run on any computer in the world (as in our planet and the spacecrafts that we sent out), as long as it is on the internet and as long as the computer allows the program to send and receive messages (via TCP). We call this arrangement a universe and the program that coordinates it all a *universe server* or just server.

This section explains what messages are, how to send them from a world program, how to receive them, and how to connect a world program to a universe.

Messages

After a world program has become a part of a universe, it may send messages and receive them. In terms of data, a message is just an S-expression.

S-expression An S-expression is roughly a nested list of basic data; to be precise, an S-expression is one of:

- a string,
- a symbol,
- a number,
- a boolean,

- a char, or
- a list of S-expressions.

Note the last clause includes `empty` of course.

```
(sexp? x) → boolean?
  x : any/c
```

determines whether `x` is an S-expression.

Sending Messages

Each world-producing callback in a world program—those for handling clock tick events, keyboard events, and mouse events—may produce a Package in addition to just a WorldState.

Package represents a pair consisting of a WorldState and a message from a world program to the server. Because programs only send messages via Package, the teachpack does not provide the selectors for the structure, only the constructor and a predicate.

```
(package? x) → boolean?
  x : any/c
```

determine whether `x` is a Package.

```
(make-package w m) → package?
  w : any/c
  m : sexp?
```

create a Package from a WorldState and an S-expression.

As mentioned, all event handlers may return WorldStates or Packages; here are the revised specifications:

```
(on-tick tick-expr)

  tick-expr : (-> WorldState (or/c WorldState package?))
```

```
(on-tick tick-expr rate-expr)

  tick-expr : (-> WorldState (or/c WorldState package?))
```

`rate-expr : natural-number/c`

`(on-key change-expr)`

`change-expr : (-> WorldState key-event? (or/c WorldState package?))`

`(on-mouse clack-expr)`

```
clack-expr : (-> WorldState natural-number/c natural-number/c MouseEvent
                 (or/c WorldState package?))
```

If one of these event handlers produces a Package, the content of the world field becomes the next world and the message field specifies what the world sends to the universe. This distinction also explains why the data definition for WorldState may not include a Package.

Connecting with the Universe

Messages are sent to the universe program, which runs on some computer in the world. The next section is about constructs for creating such a universe server. For now, we just need to know that it exists and that it is the recipient of messages.

IP `string?`

Before a world program can send messages, it must register with the server. Registration must specify the internet address of the computer on which the server runs, also known as an IP address or a host. Here a IP address is a string of the right shape, e.g., `"192.168.1.1"` or `"www.google.com"`.

`LOCALHOST : string?`

the IP of your computer. Use it while you are developing a distributed program, especially while you are investigating whether the participating world programs collaborate in an appropriate manner. This is called *integration testing* and differs from unit testing quite a bit.

A `big-bang` description of a world program that wishes to communicate with other programs must contain a `register` clause of one of the following shapes:

- `(register ip-expr)`

  `ip-expr : string?`

  connect this world to a universe server at the specified `ip-expr` address and set up capabilities for sending and receiving messages.

- ---

  `(name name-expr)`

  `name-expr : (or/c symbol? string?)`

  provide a name (`namer-expr`) to this world, which is used as the title of the canvas and the name sent to the server.

When a world program registers with a universe program and the universe program stops working, the world program stops working, too.

Receiving Messages

Finally, the receipt of a message from the server is an event, just like tick events, keyboard events, and mouse events. Dealing with the receipt of a message works exactly like dealing with any other event. DrScheme applies the event handler that the world program specifies; if there is no clause, the message is discarded.

The `on-receive` clause of a `big-bang` specifies the event handler for message receipts.

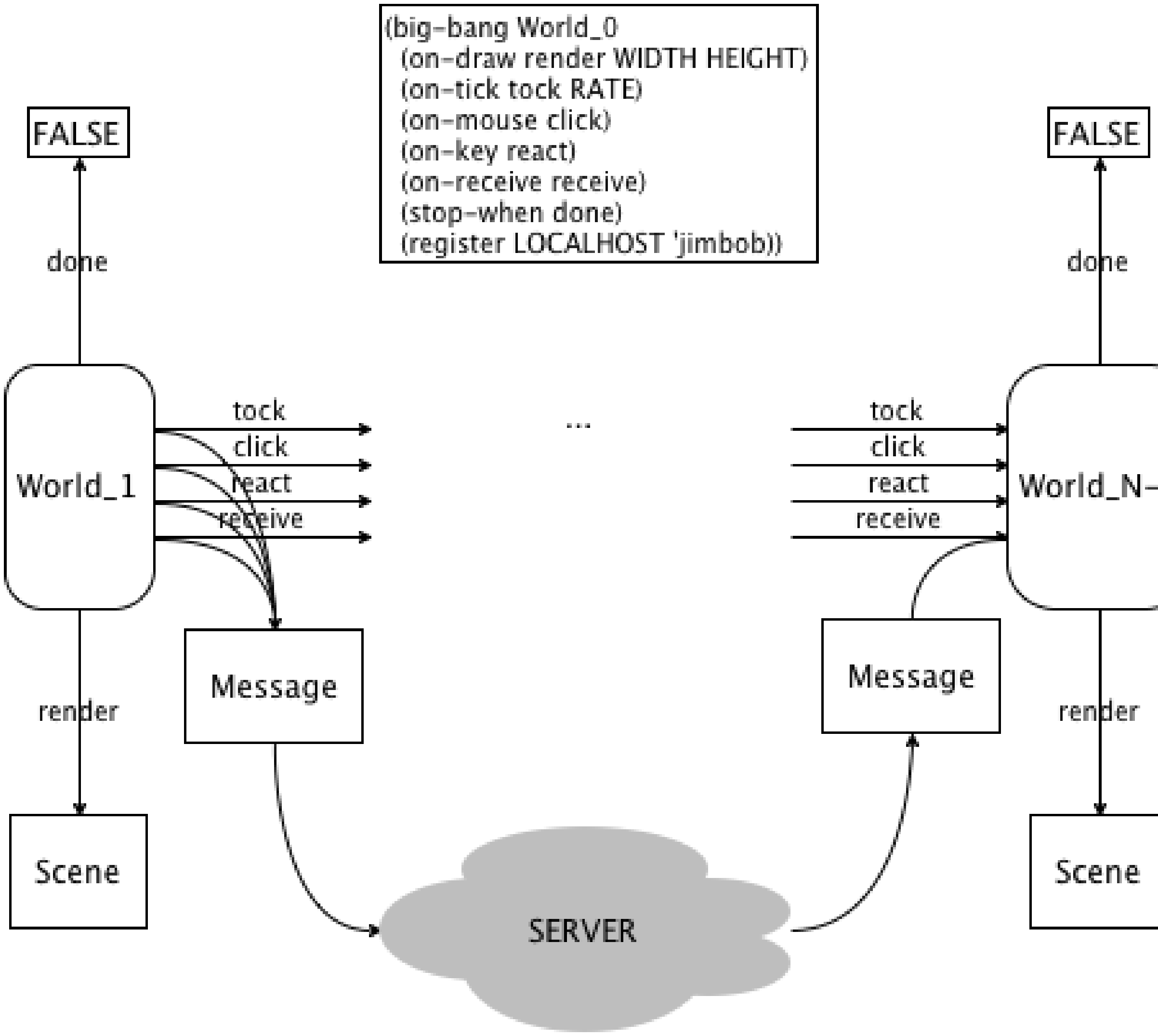
`(on-receive receive-expr)`

`receive-expr : (-> WorldState sexp? (or/c WorldState package?))`

tell DrScheme to call *`receive-expr`* for every message receipt, on the current WorldState and the received message. The result of the call becomes the current WorldState.

Because *`receive-expr`* is (or evaluates to) a world-transforming function, it too can produce a Package instead of just a WorldState. If the result is a Package, its message content is sent to the server.

The diagram below summarizes the extensions of this section in graphical form.



```
(big-bang World_0
  (on-draw render WIDTH HEIGHT)
  (on-tick tock RATE)
  (on-mouse click)
  (on-key react)
  (on-receive receive)
  (stop-when done)
  (register LOCALHOST 'jimbob))
```

A registered world program may send a message to the universe server at any time by returning a Package from an event handler. The message is transmitted to the server, which may forward it to some other world program as given or in some massaged form. The arrival of a message is just another event that a world program must deal with. Like all other event handlers *receive* accepts a WorldState and some auxiliary arguments (a message in this case) and produces a WorldState or a Package.

When messages are sent from any of the worlds to the universe or vice versa, there is no need for the sender and receiver to synchronize. Indeed, a sender may dispatch as many messages as needed without regard to whether the receiver has processed them yet. The messages simply wait in queue until the receiving server or world program take care of them.

3.1.7 The Universe Server

A *server* is the central control program of a universe and deals with receiving and sending of messages between the world programs that participate in the universe. Like a world program, a server is a program that reacts to events, though to different events than worlds. The two primary kinds of events are the appearance of a new world program in the universe and the receipt of a message from a world program.

The teachpack provides a mechanism for designating event handlers for servers that is quite similar to the mechanism for describing world programs. Depending on the designated event handlers, the server takes on distinct roles:

- A server may be a "pass through" channel between two worlds, in which case it has no other function than to communicate whatever message it receives from one world to the other, without any interference.
- A server may enforce a "back and forth" protocol, i.e., it may force two (or more) worlds to engage in a civilized tit-for-tat exchange. Each world is given a chance to send a message and must then wait to get a reply before it sends anything again.
- A server may play the role of a special-purpose arbiter, e.g., the referee or administrator of a game. It may check that each world "plays" by the rules, and it administrate the resources of the game.

As a matter of fact, a pass-through server can become basically invisible, making it appear as if all communication goes from peer world to peer in a universe.

This section first introduces some basic forms of data that the server uses to represent worlds and other matters. Second, it explains how to describe a server program.

Worlds and Messages

Understanding the server's event handling functions demands several data representations: that of (a connection to) a world program and that of a response of a handler to an event.

- The server and its event handlers must agree on a data representation of the worlds that participate in the universe.

```
(iworld? x) → boolean?
  x : any/c
```

determines whether `x` is a *iworld*. Because the universe server represents worlds via structures that collect essential information about the connections, the teachpack does not export any constructor or selector functions on worlds.

```
(iworld=? u v) → boolean?
  u : iworld?
  v : iworld?
```

compares two *iworld*s for equality.

```
(iworld-name w) → symbol?
  w : iworld?
```

extracts the name from a *iworld* structure.

```
iworld1 : iworld?
```

an *iworld* for testing your programs

```
iworld2 : iworld?
```

another iworld for testing your programs

```
iworld3 : iworld?
```

and a third one

The three sample iworlds are provided so that you can test your functions for universe programs. For example:

```
(check-expect (iworld=? iworld1 iworld2) false)
(check-expect (iworld=? iworld2 iworld2) true)
```

- Each event handler produces a *bundle*, which is a structure that contains the list of *iworld*s that the universe must track; the server's remaining state; and a list of mails to other worlds:

```
(bundle? x) → boolean?
  x : any/c
```

determines whether `x` is a *bundle*.

```
(make-bundle low state mails) → bundle?
  low : (listof iworld?)
  state : any/c
  mails : (listof mail?)
```

creates a *bundle* from a list of iworlds, a piece of data that represents a server state, and a list of mails.

If an event handler returns a bundle with an empty list of worlds, the universe server is restarted in the initial state.

A *mail* represents a message from an event handler to a world. The teachpack provides only a predicate and a constructor for these structures:

```
(mail? x) → boolean?
  x : any/c
```

determines whether `x` is a *mail*.

```
(make-mail to content) → mail?
  to : iworld?
  content : sexp?
```

creates a *mail* from a *iworld* and an S-expression.

Universe Descriptions

A server keeps track of information about the universe that it manages. One kind of tracked information is obviously the collection of participating world programs, but in general the kind of information that a server tracks and how the information is represented depends on the situation and the programmer, just as with world programs.

UniverseState `any/c` represents the server's state. For running universes, the teachpack demands that you come up with a data definition for (your state of the) server. Any piece of data can represent the state. We just assume that you introduce a data definition for the possible states and that your event handlers are designed according to the design recipe for this data definition.

The server itself is created with a description that includes the first state and a number of clauses that specify functions for dealing with universe events.

```
(universe state-expr clause ...)

clause = (on-new new-expr)
       | (on-msg msg-expr)
       | (on-tick tick-expr)
       | (on-tick tick-expr rate-expr)
       | (on-disconnect dis-expr)
       | (to-string render-expr)
```

creates a server with a given state, `state-expr`. The behavior is specified via handler functions through mandatory and optional *clause*s. These functions govern how the server deals with the registration of new worlds, how it disconnects worlds, how it sends messages from one world to the rest of the registered worlds, and how it renders its current state as a string.

Evaluating a `universe` expression starts a server. Visually it opens a console window on which you can see that worlds join, which messages are received from which world, and which messages are sent to which world. For convenience, the console also has two buttons: one for shutting down a universe and another one for re-starting it. The latter functionality is especially useful during the integration of the various pieces of a distributed program.

The mandatory clauses of a `universe` server description are `on-new` and `on-msg`:

- ---

  `(on-new new-expr)`

  `new-expr : (-> [listof iworld?]` UniverseState `iworld? bundle?)`

  tell DrScheme to call the function `new-expr` every time another world joins the universe. The event handler is called on the current list of iworlds and the joining iworld, which isn't on the list yet. In particular, the handler may reject a world program from participating in a universe, simply by not including it in the resulting `bundle` structure. The handler may still send one message to the world that attempts to join.

- ---

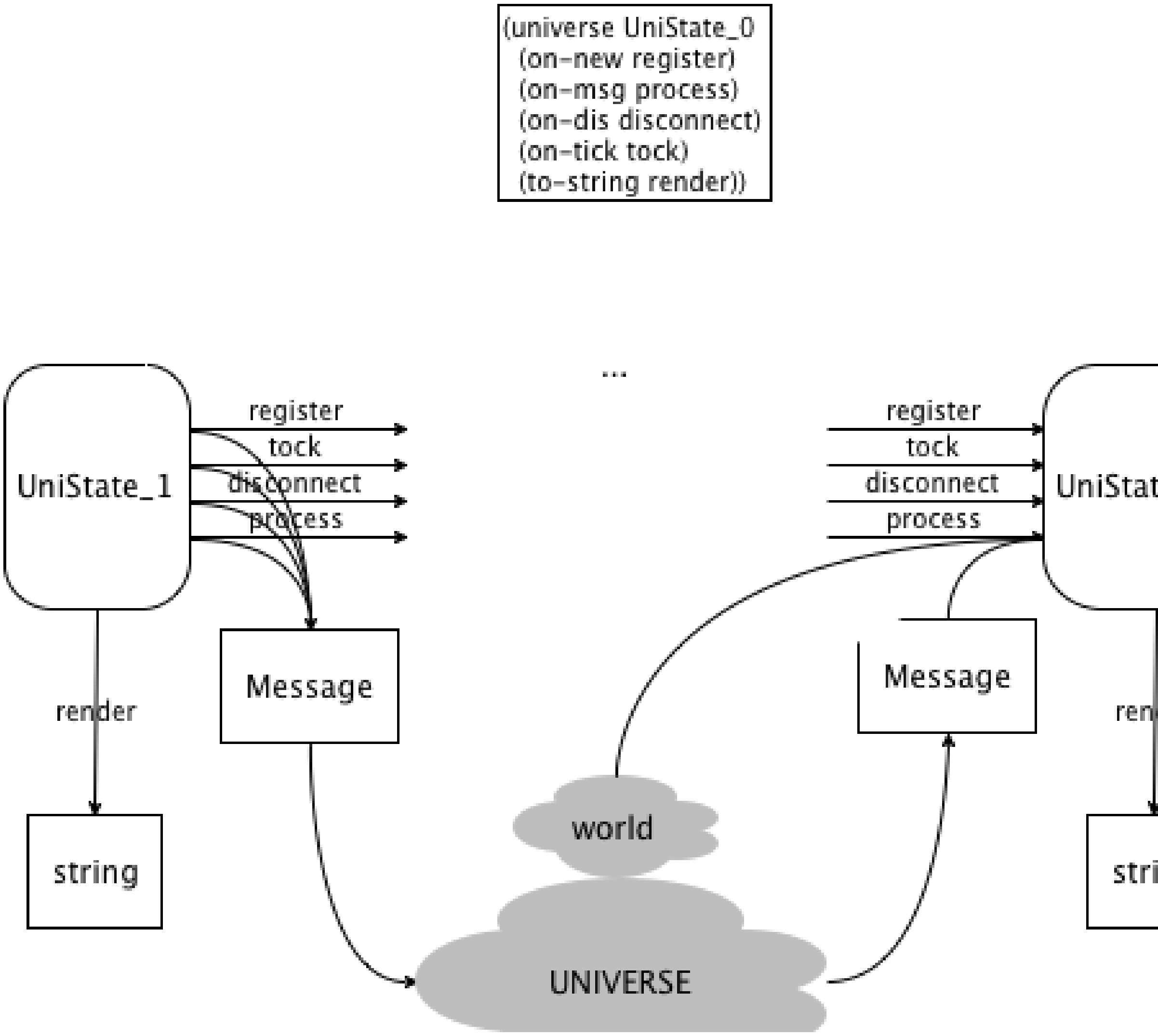
  `(on-msg msg-expr)`

  `msg-expr : (-> [listof iworld?]` UniverseState `iworld? sexp? bundle?)`

  tell DrScheme to apply `msg-expr` to the list of currently participating worlds `low`, the current state of the universe, the world `w` that sent the message, and the message itself. Note that `w` is guaranteed to be on the list `low`.

All proper event handlers produce a *bundle*. The list of worlds in this *bundle* becomes the server's list of worlds, meaning that only the server listens only to messages from ”approved” worlds. The state in the bundle is safe-guarded by the server until the next event, and the mails are broadcast as specified.

The following picture provides a graphical overview of the server's workings.



```
(universe UniState_0
  (on-new register)
  (on-msg process)
  (on-dis disconnect)
  (on-tick tock)
  (to-string render))
```

In addition to the mandatory handlers, a program may wish to add some optional handlers:

- `(on-tick tick-expr)`

  `tick-expr : (-> [listof iworld?] UniverseState bundle?)`

  tell DrScheme to apply `tick-expr` to the current list of participating worlds and the current state of the universe.

`(on-tick tick-expr rate-expr)`

`tick-expr : (-> [listof iworld?] UniverseState bundle?)`
`rate-expr : natural-number/c`

tell DrScheme to apply `tick-expr` as above but use the specified clock tick rate instead of the default.

- `(on-disconnect dis-expr)`

  `dis-expr : (-> [listof iworld?] UniverseState iworld? bundle?)`

  tell DrScheme to invoke `dis-expr` every time a participating world drops its connection to the server. The first two arguments are the current list of participating worlds and the state of the universe; the third one is the world that got disconnected.

- `(to-string render-expr)`

  `render-expr : (-> [listof iworld?] UniverseState string?)`

  tell DrScheme to render the state of the universe after each event and to display this string in the universe console.

3.1.8 A First Sample Universe

This section uses a simple example to explain the design of a universe, especially its server and some participating worlds. The first subsection explains the example, the second introduces the general design plan for such universes. The remaining sections present the full-fledged solution.

Two Ball Tossing Worlds

Say we want to represent a universe that consists of a number of worlds and that gives each world a “turn” in a round-robin fashion. If a world is given its turn, it displays a ball that ascends from the bottom of a canvas to the top. It relinquishes its turn at that point and the server gives the next world a turn.

Here is an image that illustrates how this universe would work if two worlds participated:

The two world programs could be located on two distinct computers or on just one. A server mediates between the two worlds, including the initial start-up.

Hints on Designing Universes

The first step in designing a universe is to understand the coordination of the worlds from a global perspective. To some extent, it is all about knowledge and the distribution of knowledge throughout a system. We know that the universe doesn't exist until the server starts and the worlds are joining. Because of the nature of computers and networks, however, we may assume little else. Our network connections ensure that if some world or the server sends two messages to the *same* place in some order, they arrive in the same order (if they arrive at all). In contrast, if two distinct world programs send one message each, the network does not guarantee the order of arrival at the server; similarly, if the server is asked to send some messages to several distinct world programs, they may arrive at those worlds in the order sent or in the some other order. In the same vein, it is impossible to ensure that one world joins before another. Worst, when someone removes the connection (cable, wireless) between a computer that runs a world program and the rest of the network or if some network cable is cut, messages don't go anywhere. Due to this vagaries, it is therefore the designer's task to establish a protocol that enforces a certain order onto a universe and this activity is called *protocol design*.

From the perspective of the universe, the design of a protocol is about the design of data representations for tracking universe information in the server and the participating worlds and the design of a data representation for messages. As for the latter, we know that they must be S-expressions, but usually world programs don't send all kinds of S-expressions. The data definitions for messages must therefore select a subset of suitable S-expressions. As for the state of the server and the worlds, they must reflect how they currently relate to the universe. Later, when we design their "local" behavior, we may add more components to their state space.

In summary, the first step of a protocol design is to introduce:

- a data definition for the information about the universe that the server tracks, call it UniverseState;
- a data definition for the world(s) about their current relationship to the universe;
- data definitions for the messages that are sent from the server to the worlds and vice versa. Let's call them *S2W* for messages from the server to the worlds and *W2S* for the other direction; in the most general case you may need one pair per world.

If all the worlds exhibit the same behavior over time, a single data definition suffices for step 2. If they play different roles, we may need one data definition per world.

Of course, as you define these collections of data always keep in mind what the pieces of data mean, what they represent from the universe's perspective.

The second step of a protocol design is to figure out which major events—the addition of a world to the universe, the arrival of a message at the server or at a world—to deal with and what they imply for the exchange of messages. Conversely, when a server sends a message to a world, this may have implications for both the state of the server and the state of the world. A good tool for writing down these agreements is an interaction diagram.

```
Server              World1                  World2
  |                   |                       |
  |   ’go             |                       |
  |<------------------|                       |
  |    ’go            |                       |
  |------------------------------------------>|
  |                   |                       |
  |                   |                       |
```

Each vertical line is the life line of a world program or the server. Each horizontal arrow denotes a message sent from one universe participant to another.

The design of the protocol, especially the data definitions, have direct implications for the design of event handling functions. For example, in the server we may wish to deal with two kinds of events: the joining of a new world and the receipt of a message from one of the worlds. This translates into the design of two functions with the following headers,

```
; Bundle is
; (make-bundle [Listof iworld?] UniverseState [Listof mail?])

; [Listof iworld?] UniverseState iworld? -> Bundle
; compute next list of worlds and new UniverseState
; when world w is joining the universe, which is in state s;
(define (add-world s w) ...)

; [Listof iworld?] UniverseState iworld? W2U -> Bundle
; compute next list of worlds and new UniverseState
; when world w is sending message m to universe in state s
(define (process s p m) ...)
```

Finally, we must also decide how the messages affect the states of the worlds; which of their callback may send messages and when; and what to do with the messages a world receives. Because this step is difficult to explain in the abstract, we move on to the protocol design for the universe of ball worlds.

Designing the Ball Universe

Running the ball universe has a simple overall goal: to ensure that at any point in time, one world is active and all others are passive. The active world displays a moving ball, and the passive worlds should display something, anything that indicates that it is some other world’s turn.

As for the server’s state, it must obviously keep track of all worlds that joined the universe, and it must know which one is active and which ones are passive. Of course, initially the universe is empty, i.e., there are no worlds and, at that point, the server has nothing to track.

While there are many different useful ways of representing such a universe, we just use the list of *iworlds* that is handed to each handler and that handlers return via their bundles. The UniverseState itself is useless for this trivial example. We interpret non-empty lists as those where the first *iworld* is active and the remainder are the passive *iworld*s. As for the two possible events,

- it is natural to add new *iworld*s to the end of the list; and
- it is natural to move an active *iworld* that relinquishes its turn to the end of the list, too.

The server should send messages to the first *iworld* of its list as long as it wishes this *iworld* to remain active. In turn, it should expect to receive messages only from this one active *iworld* and no other *iworld*. The content of these two messages is nearly irrelevant because a message from the server to a *iworld* means that it is the *iworld*’s turn and a message from the *iworld* to the server means that the turn is over. Just so that we don’t confuse ourselves, we use two distinct symbols for these two messages:

- A *GoMessage* is `'it-is-your-turn`.
- A *StopMessage* is `'done`.

From the universe’s perspective, each world is in one of two states:

- A passive world is *resting*. We use `'resting` for this state.
- An active world is not resting. We delay choosing a representation for this part of a world’s state until we design its “local” behavior.

It is also clear that an active world may receive additional messages, which it may ignore. When it is done with its turn, it will send a message.

```
Server
  |                 World1
  |<==================|
  |  'it-is-your-turn |
  |------------------>|
  |                   |                    World2
```

```
|<==========================================|
|  'done            |                       |
|<------------------|                       |
|  'it-is-your-turn |                       |
|------------------------------------------>|
|                   |                       |
|                   |                       |
|  'done            |                       |
|<------------------------------------------|
|  'it-is-your-turn |                       |
|------------------>|                       |
|                   |                       |
|                   |                       |
```

Here the double-lines (horizontal) denote the registration step, the others are message exchanges. The diagram thus shows how the server decides to make the first registered world the active one and to enlist all others as they join.

Designing the Ball Server

The preceding subsection dictates that our server program starts like this:

```
; teachpack: universe.ss

; UniverseState is '*
; StopMessage is 'done.
; GoMessage is 'it-is-your-turn.
```

The design of a protocol has immediate implications for the design of the event handling functions of the server. Here we wish to deal with two events: the appearance of a new world and the receipt of a message. Based on our data definitions and based on the general contracts of the event handling functions spelled out in this documentation, we get two functions for our wish list:

```
; Result is
; (make-bundle [Listof world?] '* (list (make-mail world? GoMessage)))

; [Listof world?] UniverseState world? -> Result
; add world w to the universe, when server is in state u
(define (add-world u w) ...)

; [Listof world?] UniverseState world? StopMessage -> Result
; world w sent message m when server is in state u
(define (switch u w m) ...)
```

Although we could have re-used the generic contracts from this documentation, we also know from our protocol that our server sends a message to exactly one world. Note how these contracts are just refinements of the generic ones. (A type-oriented programmer would say that the contracts here are subtypes of the generic ones.)

The second step of the design recipe calls for functional examples:

```
; an obvious example for adding a world:
(check-expect
  (add-world '() '* world1)
  (make-bundle (list world1)
               '*
               (list (make-mail world1 'it-is-your-turn))))

; an example for receiving a message from the active world:
(check-expect
 (switch (list world1 world2) '* world1 'done)
 (make-bundle (list world2 world1)
              '*
              (list (make-mail world2 'it-is-your-turn))))
```

Note that our protocol analysis dictates this behavior for the two functions. Also note how we use `world1`, `world2`, and `world3` because the teachpack applies these event handlers to real worlds.

Exercise: Create additional examples for the two functions based on our protocol.

The protocol tells us that *add-world* just adds the given *world* structure—recall that this a data representation of the actual world program—to the given list of worlds. It then sends a message to the first world on this list to get things going:

```
(define (add-world univ state wrld)
  (local ((define univ* (append univ (list wrld))))
    (make-bundle univ*
                 '*
                 (list (make-mail (first univ*) 'it-is-your-turn)))))
```

Because *univ** contains at least *wrld*, it is acceptable to create a mail to `(first univ*)`. Of course, this same reasoning also implies that if *univ* isn't empty, its first element is an active world and is about to receive a second `'it-is-your-turn` message.

Similarly, the protocol says that when *switch* is invoked because a world program sends a message, the data representation of the corresponding world is moved to the end of the list and the next world on the (resulting) list is sent a message:

```
(define (switch univ state wrld m)
  (local ((define univ* (append (rest univ) (list (first univ)))))
```

```
    (make-bundle univ*
                 '*
                 (list (make-mail (first univ*) 'it-is-your-turn)))))
```

As before, appending the first world to the end of the list guarantees that there is at least this one world on this list. It is therefore acceptable to create a mail for this world.

Start the server now.

```
(universe '* (on-new add-world) (on-msg switch))
```

Exercise: The function definition simply assumes that *wrld* is `world=?` to `(first univ)` and that the received message *m* is `'done`. Modify the function definition so that it checks these assumptions and raises an error signal if either of them is wrong. Start with functional examples. If stuck, re-read the section on checked functions from HtDP. (Note: in a universe it is quite possible that a program registers with a server but fails to stick to the agreed-upon protocol. How to deal with such situations properly depends on the context. For now, stop the universe at this point by returning an empty list of worlds. Consider alternative solutions, too.)

Exercise: An alternative state representation would equate UniverseState with *world* structures, keeping track of the active world. The list of world in the server would track the passive worlds only. Design appropriate `add-world` and `switch` functions.

Designing the Ball World

The final step is to design the ball world. Recall that each world is in one of two possible states: active or passive. The second kind of world moves a ball upwards, decreasing the ball's *y* coordinate; the first kind of world displays something that says it's someone else's turn. Assuming the ball always moves along a vertical line and that the vertical line is fixed, the state of the world is an enumeration of two cases:

```
; teachpack: universe.ss

; WorldState is one of:
; -- Number          %% representing the y coordinate
; -- 'resting

(define WORLD0 'resting)

; A WorldResult is one of:
; -- WorldState
; -- (make-package WorldState StopMessage)
```

The definition says that initially a world is passive.

The communication protocol and the refined data definition of WorldState imply a number of contract and purpose statements:

```
; WorldState GoMessage -> WorldResult
; make sure the ball is moving
(define (receive w n) ...)

; WorldState -> WorldResult
; move this ball upwards for each clock tick
; or stay 'resting
(define (move w) ...)

; WorldState -> Scene
; render the world as a scene
(define (render w) ...)
```

Let's design one function at a time, starting with *receive*. Since the protocol doesn't spell out what *receive* is to compute, let's create a good set of functional examples, exploiting the structure of the data organization of WorldState:

```
(check-expect (receive 'resting 'it-is-your-turn) HEIGHT)
(check-expect (receive (- HEIGHT 1) 'it-is-your-turn) ...)
```

Since there are two kinds of states, we make up at least two kinds of examples: one for a `'resting` state and another one for a numeric state. The dots in the result part of the second unit test reveal the first ambiguity; specifically it isn't clear what the result should be when an active world receives another message to activate itself. The second ambiguity shows up when we study additional examples, which are suggested by our approach to designing functions on numeric intervals (HtDP, section 3). That is we should consider the following three inputs to *receive*:

- `HEIGHT` when the ball is at the bottom of the scene;
- `(- HEIGHT 1)` when the ball is properly inside the scene; and
- `0` when the ball has hit the top of the scene.

In the third case the function could produce three distinct results: `0`, `'resting`, or `(make-package 'resting 'done)`. The first leaves things alone; the second turns the active world into a resting one; the third does so, too, and tells the universe about this switch.

We choose to design *receive* so that it ignores the message and returns the current state of an active world. This ensures that the ball moves in a continuous fashion and that the world remains active.

Exercise: One alternative design is to move the ball back to the bottom of the scene every time `'it-is-your-turn` is received. Design this function, too.

```
(define (receive w m)
  (cond
    [(symbol? w) HEIGHT] ; meaning: (symbol=? w 'resting)
    [else w]))
```

Our second function to design is *move*, the function that computes the ball movement. We have the contract and the second step in the design recipe calls for examples:

```
; WorldState -> WorldState or (make-package 'resting 'done)
; move the ball if it is flying

(check-expect (move 'resting) 'resting)
(check-expect (move HEIGHT) (- HEIGHT 1))
(check-expect (move (- HEIGHT 1)) (- HEIGHT 2))
(check-expect (move 0) (make-package 'resting 'done))

(define (move x) ...)
```

Following HtDP again, the examples cover four typical situations: `'resting`, two end points of the specified numeric interval, and one interior point. They tell us that *move* leaves a passive world alone and that it otherwise moves the ball until the y coordinate becomes `0`. In the latter case, the result is a package that renders the world passive and tells the server about it.

Turning these thoughts into a complete definition is straightforward now:

```
(define (move x)
  (cond
    [(symbol? x) x]
    [(number? x) (if (<= x 0) (make-package 'resting 'done) (sub1 x))]))
```

Exercise: what could happen if we had designed *receive* so that it produces `'resting` when the state of the world is `0`? Use your answer to explain why you think it is better to leave this kind of state change to the tick event handler instead of the message receipt handler?

Finally, here is the third function, which renders the state as a scene:

```
; WorldState -> Scene
; render the state of the world as a scene

(check-expect (render HEIGHT) (place-image BALL 50 HEIGHT MT))
(check-expect (render 'resting)
              (place-image  (text "resting" 11 'red) 10 10 MT))

(define (render w)
  (place-image
    (text name 11 'black) 5 85
```

```
    (cond
      [(symbol? w) (place-image (text "resting" 11 'red) 10 10 MT)]
      [(number? w) (place-image BALL 50 w MT)])))
```

Here is an improvement that adds a name to the scene and abstracts over the name at the same time:

```
; String -> (WorldState -> Scene)
; render the state of the world as a scene

(check-expect
 ((draw "Carl") 100)
 (place-image (text "Carl" 11 'black)
              5 85
              (place-image BALL 50 100 MT)))

(define (draw name)
  (lambda (w)
    (place-image
     (text name 11 'black) 5 85
     (cond
       [(symbol? w) (place-image (text "resting" 11 'red) 10 10 MT)]
       [(number? w) (place-image BALL 50 w MT)]))))
```

By doing so, we can use the same program to create many different worlds that register with a server on your computer:

```
; String -> WorldState
; create and hook up a world with the LOCALHOST server
(define (create-world n)
  (big-bang WORLD0
            (on-receive receive)
            (on-draw (draw n))
            (on-tick move)
            (name n)
            (register LOCALHOST)))
```

Now you can use `(create-world 'carl)` and `(create-world 'same)`, respectively, to run two different worlds, after launching a server first.

Exercise: Design a function that takes care of a world to which the universe has lost its connection. Is *Result* the proper contract for the result of this function?