Extended Exercise: Interactive Games, v207

A Supplement to "How to Design Programs"

1 Events

When something interesting happens, people say an event takes place. Programs also notice events. When people press keys on a computer or when the clock ticks, an event happens and programs react to it. We all know such programs.

Text editing programs interpret every keystroke. Some indicate that the user is typing plain text; others request that the editor changes fonts, transposes words, and so on. Game programs also heavily use arrow keys; they allow players to move objects, fire weapons, open locks, and so on. Games and animation programs also react to the ticking of the clock. Every time it ticks, they must redraw some canvas.

1.1 Interaction via Keystrokes

PREREQUISITE: 7. The Varieties of Data
TEACHPACK: draw.ss

The **draw.ss** teachpack supports programs that react to events on a keyboard. Every keystroke is an event. DrScheme observes these events and provides a function that delivers the keystrokes, if the user has touched the keyboard. Clearly, the first thing we need then is a data definition that describes this class of events:[1]

```
;; A KeyEvent is one of:
;; — a Character, e.g., #\a, #\space,
;; — a Symbol, e.g., 'up, 'left,
```

A character represents the event when the user presses an alphanumeric key; a symbols represents special keys and keyboard events.

[1]Recall that char? is the predicate that recognizes characters and that char=? is a predicate for comparing characters.

The data definition puts together an interesting mix of distinct classes of data into one class. Naturally, a function that consumes and processes a *KeyEvent* must distinguish all those cases.

Now consider the following problem:

> Develop a program that draws the movements of a ball on a straight line left or right. The user controls the movement of the ball by pressing the left arrow (←) key or the right arrow (→) key on the keyboard.

Clearly, one of the ingredients that this program needs is a function that consumes the current *x* coordinate of the ball and a key event and that produces the new *x* coordinate:

```
;; ball-move : KeyEvent Number → Number
;; to interpret a 'left or 'right keystroke as a move
;; into the appropriate direction by 10 pixels
```

Since keyevents are ordinary pieces of data, we can also make up some example:

```
(ball-move #\a 0) "should be" 0
(ball-move #\z 20) "should be" 20
(ball-move 'left 0) "should be" −10
(ball-move 'right 20) "should be" 30
(ball-move 'up 0) "should be" 0
```

The examples illustrate that the function must distinguish character key events from symbolic ones and that it must distinguish 'left and 'right from other symbolic events.

Producing a template from the data definition and a full function from the template and the examples is now straightforward:

```
;; ball-move : KeyEvent Number → Number
;; to interpret a 'left or 'right keystroke as a move
;; into the appropriate direction by 10 pixels
(define (ball-move ke x-ball)
  (cond
    [(char? ke) x-ball]
    [else ; we now know that (symbol? ke) is true
      (cond
        [(symbol=? 'left ke) (− x-ball 10)]
        [(symbol=? 'right ke) (+ x-ball 10)]
        [else x-ball])]))
```

```
;; World = Ball
;; Ball = Number

;; KeyEvent World → World
;; erase the existing ball, move it, and draw it again
(define (erase-and-draw ke x-ball)
  (draw (ball-erase x-ball)
        (ball-paint (ball-move ke x-ball))
        produce
        (ball-move ke x-ball)))

;; KeyEvent Ball → Ball
;; move the ball according to ke
(define (ball-move ke x-ball)
  (cond
    [(char? ke) x-ball]
    ;; we now know that (symbol? ke) is true
    [(symbol=? 'left ke) (- x-ball 10)]
    [(symbol=? 'right ke) (+ x-ball 10)]
    [else x-ball]))

;; Ball → true
;; paint the ball from the canvas
(define (ball-paint x-ball) (draw-solid-disk (make-posn x-ball 50) 10 'red))

;; Ball → true
;; earse the ball from the canvas
(define (ball-erase x-ball) (clear-solid-disk (make-posn x-ball 50) 10 'red))

;; run program run
(define ball0 150)

(start 300 100) ;; create the canvas
(big-bang 1 ball0) ;; specify what the initial world is
(ball-paint ball0) ;; paint the initial world on the canvas
(on-key-event erase-and-draw) ;; specify how to deal with key events
```

Figure 1: Moving a ball

The **cond** expression distinguishes the two subclasses of data in the data definition; the nested **cond** recognizes the kind of key that the user pressed. Of course, we can also write this more concisely with a single **cond**-expression as the definition in figure 1 shows. The comment line reminds us that we first distinguish two major subclasses and then cases within this subclass.

Of course, *ball-move* doesn't really move any ball on any canvas. It only computes where the ball should appear next on the horizontal line, given the user's keystroke and the ball's current position. To accomplish this, we need a function that uses *ball-move*, paints, and erases the ball from the canvas at the proper time. The function *erase-and-draw* in figure 1 accomplishes just that. Like *ball-move*, the function consumes a key event and a ball. It then erases the existing ball from the canvas, computes where the ball is supposed to be next, paints it there, and returns this new ball representation.

To understand how these functions produce the desired effects, we need to study three more pieces of the `draw.ss` teachpack:

1. *big-bang* is a function that consumes a number t and a *World*. It starts the clock, make it tick every t seconds, and sets the world to be w. A *World* is just a piece of data.

   In our example, the *World* is just the x coordinate of the ball. After all, from that we know where the ball is, because it moves only on a vertical line. The bottom of figure 1 therefore means that we first create the canvas and the world and then draw the world.

2. (**draw** *draw-command* ... **produce** *e*) executes a series of drawing command and then returns the value of the expression *e*. Putting the previous point and this one together explains why *erase-and-draw* erases the current "world" from the canvas and then draw the new one.

3. *on-key-event* consumes a function like *erase-and-draw* and attaches it to the keyboard. We call such functions KEY EVENT HANDLER.

   When the player touches a key, DrScheme applies *erase-and-draw* to an appropriate character or symbol and the current world. In turn, the world that *erase-and-draw* produces becomes the current world and thus the argument for the next application of the function.

Exercises

Exercise 1.1.1 Copy and paste the program from figure 1 into DrScheme. Use it to move the ball around. ∎

Exercise 1.1.2 Design the function *up-or-down*, which consumes a *KeyEvent* and produces true when the input is 'up, 'down, #\u, or #\d. ∎

Exercise 1.1.3 Design *move-4-directions*. The function consumes a *KeyEvent* and produces a *Posn*. The latter represents how far the ball on the canvas has to move in one of the four directions ('left, 'right, 'up, 'down). ∎

1.2 Time

The **draw.ss** teachpack also supports programs that must react to the ticking of the clock. For example, if a program is supposed to move a ball continuously across the screen, it really means that for each tick of the clock the program erases the ball from its current position and draws it at the new position.

Let us see how to solve this programming problem:

> Develop a program that draws the movements of a ball as it moves from left to right across a canvas in a straight line. The clock controls the movement of the ball, i.e., the ball should move every 1/10 of a second.

The problem basically says that the program should apply some function like *erase-and-draw* from figure 1 every 1/10 of a second. Put differently, we need a function that deals with time events but is otherwise just like a function that deals with keyboard events.

For just such problems, the `draw.ss` teachpack provides two more functions:

4. *on-tick-event*, which consumes a function that is applied every tick of the clock. The function is a TIME EVENT HANDLER. In contrast to a key event handler, it consumes only a representation of the world not other value; its result is also a *World*.

5. *end-of-time*, which stops the time and produces the last *World*.

Hence, just as in the section on key events, we must first agree on what represents the world and then we can design event handlers that deal with this *World*.

Figure 2 displays the code for the specified animation. As before, our world consists of just one ball and, to be more precise, of its x coordinate, which is a number. Everything else remains as the ball moves across the canvas. Computing the new coordinate of the ball becomes much simpler.

```
;; World = Ball
;; Ball = Number

;; World → World
;; erase the existing ball, move it and draw it again
(define (erase-and-draw x-ball)
  (draw (ball-erase x-ball)
        (ball-paint (ball-move x-ball))
        produce
        (ball-move x-ball)))

;; Ball → Ball
;; move the ball right by 1 pixel
(define (ball-move x-ball) (+ x-ball 1))

;; Ball → true
;; paint the ball from the canvas
(define (ball-paint x-ball) (draw-solid-disk (make-posn x-ball 50) 10 'red))

;; Ball → true
;; earse the ball from the canvas
(define (ball-erase x-ball) (clear-solid-disk (make-posn x-ball 50) 10 'red))

;; run program run
(define ball0 150)

(start 300 100)
(big-bang .1 ball0)
(ball-paint ball0)
(on-tick-event erase-and-draw)
```

Figure 2: A simple animation

Since it doesn't depend on a keystroke anymore, our new *ball-move* function just adds 1 to the current value of the *x* coordinate. This represents a move to the right. The other functions are the same as before.

Exercises

Exercise 1.2.1 Copy and paste the program from figure 2 into DrScheme. Watch the ball move by itself. ∎

Exercise 1.2.2 Develop a program that simulates the drop of a ball from the top of the canvas to the bottom of the canvas. Represent the world as a structure:

```
(define-struct world (t y))
;; World = (make-world Number Number)
```

The first number in the structure represents the number of times that the handler has been invoked since *big-bang*, and the second one represents the current y coordinate of the ball. The ball's position is determined by the formula:

$$y = \frac{1}{2} \cdot 5 \cdot t^2$$

where t is the current time. Hint: Try (make-world 0 0) as the initial world. What does it represent? What should the next world look like? ∎

Teaching Note: The goal is to keep the functional portion separated from the drawing and event-based portion of the program. To this end, we use a program design discipline that separates the computations from the visible actions. The discipline is often referred to as model-view separation. Indeed, the idea of programming in DrScheme is based on this idea, too. DrScheme provides rudimentary views and students can focus on writing the functions proper. ∎

2 Stopping a UFO

The goal of this extended exercise is to develop a simple interactive game. Imagine the approach of a UFO, falling out of the blue sky. You're riding a modern AUP (anti-UFO platform), and your task is to stop the UFO from crossing the line, i.e., the bottom of the canvas. Your powerful AUP can move left or right, and it can shoot at the UFO in straight lines. The means of last resort is to make sure the UFO crashes into your AUP. If the UFO makes it across the line, you lost; otherwise you win.

Figure 3 contains a screen shot of the game. The (green) saucer on the canvas is the UFO; the line at the bottom is your AUP. The straight lines going up and through the UFO are the shots that the AUP fired.

The section consists of three subsections. Each corresponds to a stage in the design process. The subsections illustrate what we call the iterative refinement process (see section 16). The goal of iterative refinement is to implement the core functionality of a product and to add pieces of the functionality step by step. Here we present the method via an example;

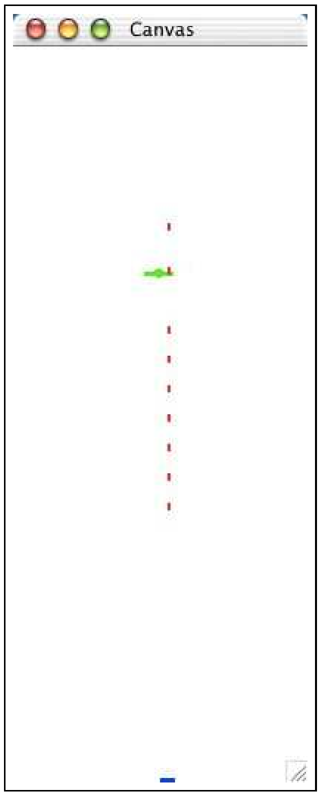

Figure 3: Stopping a UFO

take a look at the sections on iterative refinement in *How to Design Programs* for a thorough description of the idea.

2.1 UFOs

PREREQUISITE: 6.1 Structures
TEACHPACK: draw.ss

The first goal is to create a UFO that drops from the top of a canvas to the bottom. The function that does the repetitive work and the fragment that starts the game are defined in figure 4. The following exercises show how to fill in the dots.

We adopt the usual conventions from physics and think of the UFO as just a position on the canvas:

```
;; World = UFO

;; UFO → Boolean
;; fly UFO until it lands on bottom
(define (fly-until-down ufo)
  (and (start WIDTH HEIGHT)
       (big-bang .1 ufo)
       (ufo-draw ufo)
       (on-tick-event new-scene)))

;; World → World
;; erase old scene, create new one, draw it and produce it.
(define (new-scene w)
  (ufo-erase-and-draw w (ufo-move w)))

;; UFO UFO → UFO
;; erase old ufo and draw new one, unless it's at the bottom
(define (ufo-erase-and-draw old-ufo new-ufo)
  (cond
    [(ufo-at-bottom? new-ufo) (end-of-time)]
    [else (draw (ufo-clear old-ufo)
                (ufo-draw new-ufo)
                produce
                new-ufo)]))
...
;; Constants
(define WIDTH 200)
(define HEIGHT 500)

;; run program, run
(fly-until-down (ufo-create (random WIDTH)))
```

Figure 4: Flying a UFO

```
;; A UFO is a Posn: UFO = Posn
```

When we draw the UFO we think of it as a green disk whose center is the *Posn* with which we represent it. Or, we think of it as something more elaborate, but for now this doesn't matter. We can always change our understanding; we just need to keep in mind what the *Posn* represents. To remind ourselves of this relationship (between data and information), we call the *Posn* that represents a UFO an *anchor point*.

Exercises

Exercise 2.1.1 Design the function *ufo-create*, which consumes a number *n* and produces a UFO whose anchor point is at the top of the canvas *n* pixels to the right of the canvas origin. ∎

Exercise 2.1.2 Design the function *ufo-move*, which consumes a *UFO* (representation) and produces one whose anchor point is 3 pixels below the given one.

Modify *ufo-move* so that it produces a *UFO* that has dropped 3 pixels and has moved randomly to the left or right by up to 4 pixels. Hint: Use the function random, which consumes a positive number *n* and produces a number between 0 (inclusive) and *n* (exclusive). Two consecutive calls may or may not produce the same number. Design the function *random-range*, which consumes *n* and produces a number between $-n$ and $+n$.

Challenge: Revise *ufo-move-ran* so that a *UFO* that has disappeared on the left or right of the canvas reappears on the other side for the next time slice. Modify the program in figure 4 so that it uses *ufo-move-ran*. ∎

Exercise 2.1.3 Design the function *ufo-at-bottom?*, which consumes a *UFO* (representation) and determines whether its anchor point is level with, or below, the bottom of the canvas. ∎

Exercise 2.1.4 Design the function *ufo-draw*, which consumes the representation of a ufo and draws it on the canvas.

Also design *ufo-clear*, which consumes the representation of a ufo and clears it from the canvas.

Draw the UFO as a green disk of radius 3 around the anchor point. For the ambitious, draw the UFO as a green line of length 20 with a disk of radius 3 in the center. ∎

Now watch the UFO fly down from top to bottom (in a random walk).

2.2 AUPs

PREREQUISITE: 7.1 Varieties of Data
Now it's time to develop our defenses. An AUP defends the bottom of the canvas where it can move left or right.

Exercise 2.2.1 Develop a data definition for AUPs. Keep in mind that an AUP is always at the bottom of the canvas, and that it always has the same shape. ∎

```
;; World = AUP

;; AUP → Boolean
;; create a world with an AUP and install an event handler for it
(define (controller aup)
  (and (start WIDTH HEIGHT)
       (big-bang 1 aup)
       (aup-draw aup)
       (on-key-event new-aup-scene)))

;; KeyEvent World → World
;; erase old scene, create new one, draw it and produce it.
(define (new-aup-scene ke w)
  (aup-erase-and-draw w (aup-move ke w)))

;; AUP AUP → AUP
(define (aup-erase-and-draw old-aup new-aup)
  (draw (aup-clear old-aup)
        (aup-draw new-aup)
        produce
        new-aup))

...

;; Constants
(define WIDTH 200)
(define HEIGHT 500)

;; run program, run
(controller (create-aup 0))
```

Figure 5: Controlling an AUP

Exercise 2.2.2 Design the function *aup-create*, which consumes a number *n* and produces an *AUP* that is *n* pixels to the right of the canvas's left margin. ∎

Exercise 2.2.3 Design *aup-move*. The function consumes an *AUP* (representation) and a *KeyEvent*. It produces an *AUP* that has moved to the left or right by one (1) pixel, if the player has pressed the left or right arrow key; otherwise, it just returns the given *AUP*. ∎

Exercise 2.2.4 Design the function *aup-draw*, which consumes a *AUP* and draws it on the canvas.

Also design *aup-clear*, which consumes a *AUP* and clears it from the canvas.

Think of the AUP as a blue line of width 10. ∎

The program fragment in figure 5 allows players to control an *AUP* via the arrow keys on the keyboard. It requires well-developed solutions for all the exercises in this subsection. Don't rush. Follow the design recipe.

2.3 Putting it all together

PREREQUISITE: 7.1 Varieties of Data

With UFOs and AUPs in place we can create our first interactive game. First, we need to merge the code from figures 4 and 5. That gives us a world in which a UFO descends from the top and the AUP is movable via the arrow keys.

Figure 6 contains the result of the merger. The world is now a structure with two fields: one for the UFO and one for the AUP. The *game* function sets up two handlers: one for the keyboard events and one for the time events. The remaining functions are from the previous programs, except that all those whose contract contains *World* need some adjustments.

The other major addition in figure 6 concerns the end of the game. In the original code for flying a UFO, the animation just stopped when the UFO reached the ground. Now we should announce the result of the game. After all, the player can win or lose. To this end we apply the new *announcement* function to the current world. If the UFO crashes on the AUP, the player wins; otherwise, it's a loss.

Exercises

Exercise 2.3.1 Design the function *landed-on-aup?*. It determines whether some given *UFO* has landed on a given *AUP*.

Hint: Recall the geometric interpretation that goes with each data representation. Then draw pictures and determine what it means for the UFO shape to overlap with the AUP shape. If you can't get it completely right, approximate what it means for the two shapes to overlap as best as you can. Of course, you must then expect to see these simplifications as you play the game. ∎

```
(define-struct world (ufo aup))
;; World = (make-world UFO AUP)

;; UFO AUP → true
;; set up world with UFO and AUP and add event handlers
(define (game ufo aup)
  (and (start WIDTH HEIGHT)
       (big-bang .1 (make-world ufo aup))
       (ufo-draw ufo)
       (aup-draw aup)
       (on-key-event new-aup-scene)
       (on-tick-event new-scene)))

;; KeyEvent World → World
(define (new-aup-scene ke w)
  (make-world
    (world-ufo w)
    (aup-erase-and-draw (world-aup w) (aup-move ke (world-aup w)))))

;; AUP AUP → AUP
(define (aup-erase-and-draw old-aup new-aup)
  (draw (aup-clear old-aup)
        (aup-draw new-aup)
        produce
        new-aup))

;; World → World
(define (new-scene w)
  (make-world
    (ufo-erase-and-draw (world-ufo w) (ufo-move (world-ufo w)))
    (world-aup w)))

;; UFO UFO → UFO
(define (ufo-erase-and-draw old-ufo new-ufo)
  (cond
    [(ufo-at-bottom? new-ufo) (announcement old-ufo (end-of-time))]
    [else (draw (ufo-clear old-ufo)
                (ufo-draw new-ufo)
                produce
                new-ufo)]))
```

Figure 6: Crashing a UFO

Exercise 2.3.2 Design the function *announcement*. It consumes a *World* and produces `true`. If the UFO in the *World* landed on the AUP in the *World*, it draws the string `"you win"` on the canvas; otherwise, it writes `"you lose"`.

Hint: Which function in the game code applies *announcement* to the *World* and which world is it? Simulate such worlds for the tests. ∎

2.4 One shot at the UFO

PREREQUISITE: 7.1 Varieties of Data
The chief engineer has figured out how to let AUP's fire a gun—once. This means that an AUP now has two chances to stop the UFO. Either the AUP shoots and that one shot hits the UFO or it manages to stop the UFO via a crash. We already have a program that does the latter; let's develop a program that simulates the new ability to shoot.

Exercises

Exercise 2.4.1 Develop a data definition for representing a shot. Make up examples and show what each example means in figure 3. ∎

Exercise 2.4.2 Design the function *shot-create*, which consumes an *AUP* and produces (the representation of) a shot that has just left the AUP. Assume the shot leaves from the middle of the AUP. ∎

Exercise 2.4.3 Design the function *shot-move*, which consumes a *Shot* (representation) and produces one that has risen 5 pixels. ∎

Exercise 2.4.4 Design the function *shot-draw*, which consumes the representation of a shot and draws it on the canvas.

Also design *shot-clear*, which consumes the representation of a shot and clears it from the canvas.

Think of a shot as a vertical red line of length 5. ∎

Now that we have functions for creating, drawing, erasing, and moving shots, we can turn our attention to the code that drives the animation. Let's inspect each element of figure 6 and see how we need to adapt it:

1. The first element defines the *World* as a structure of two components: a UFO and an AUP. Naturally, we need to add a field in order to keep track of the shot that the AUP can fire:

   ```
   (define-struct world (ufo aup shot))
   ```

```
;; UFO AUP → true
;; set up world with UFO and AUP and add event handlers
(define (game ufo aup) ...)

;; KeyEvent World → World
(define (new-aup-scene ke w)
  (cond
    [(char? ke) w]
    [(symbol=? 'up ke)
     (cond
       [(boolean? (shot-world w))
        (make-world (world-ufo w) (world-aup w) (shot-create (world-aup w)))]
       [else w])]
    [else (make-world (world-ufo w)
                (aup-erase-and-draw (world-aup w) (aup-move ke (world-aup w)))
                (world-shot w))]))

;; AUP AUP → AUP
;; erase old AUP, draw new one, and produce it
(define (erase-and-draw aup-old aup-new) ...)

;; World → World
(define (new-scene w)
  (cond
    [(boolean? (world-shot w))
     (make-world
       (ufo-erase-and-draw (world-ufo w) (ufo-move (world-ufo w)))
       (world-aup w)
       false)]
    [(hit-shot? (world-shot w) (world-ufo w)) (announcement (end-of-time))]
    [else (make-world
                (ufo-erase-and-draw (world-ufo w) (ufo-move (world-ufo w)))
                (world-aup w)
                (shot-erase-and-draw (world-shot w) (shot-move (world-shot w))))]))

;; UFO UFO → UFO
;; erase old ufo and draw new one, unless it's at the bottom
(define (ufo-erase-and-draw old-ufo new-ufo) ...)
```

Figure 7: Crashing or shooting down a UFO

Although it is tempting to say that (make-world *UFO AUP Shot*) constructs a world, it is also clearly incorrect. Initially no shot has been fired, and we need to leave the timing of the shot to the player. Hence, we define the class of *World*s as follows:

```
(define-struct world (ufo aup shot))
;; World = (make-world UFO AUP Shot/f)

;; A Shot/f is one of the following:
;; — a shot ; (see exercise 2.4.1)
;; — false
```

In other words, as long as no shot has been fired, the *shot* field is false; afterwards, it contains the shot.

2. The second element in figure 6 is an event handler for time events. It consumes a world, erases it, computes new values for those pieces that move with each tick event, and draws those. Its result is the new world. Since the definition of what a world is has changed, we must also modify this event handler.
3. The third and last critical element in figure 6 is the key event handler. Like the time event handler, it consumes a world and produces one. The modified handler must not only check for left and right arrow keys, but also for up (↑), which we take as a signal to fire the single shot in the AUP.

Turning these thoughts into code yields the definitions in figure 7. The following exercises specify the purpose of the underlined functions. The others remain the same.

Exercises

Exercise 2.4.5 Formulate purpose statements for the functions *new-scene* and *new-aup-scene* compute. Then explain *how* they compute their results. ∎

Exercise 2.4.6 Design the function *hit-shot?*, which determines whether a shot has hit a UFO. The function consumes an *Shot* and a *UFO*. It produces true if there is any overlap between the *UFO* and *Shot*. Hint: Recall the geometric interpretation that goes with each data representation. Then draw pictures and determine what it means to figure out whether the two geometric shapes overlap. Approximate “hit by a shot” as best as you can; expect to see these simplifications as you play the game. ∎

Exercise 2.4.7 Modify *announcement* so that it writes "you win" on the canvas when the UFO is hit by a shot or when it crashes into the AUP. ∎

Exercise 2.4.8 Design *shot-erase-and-draw*. The function consumes the old shot and the new shot. It erases the old one, draws the new one, and produces it as a result. ∎

2.5 Many Shots

PREREQUISITE: 10.2 Lists that Contain Structures
The true goal is to simulate an AUP-UFO fight like the one in figure 3. The AUP in the figure can obviously fire many shots, not just one. Since "many" clearly means "arbitrary" and "unknown", we need a list of shots. Put differently, we need to revise our structure definition and our data definition for *World*:

```
(define-struct world (ufo aup shots))
;; World = (make-world UFO AUP LoShots)
```

and we must revise all the functions that deal with *World*s.

Exercises

Exercise 2.5.1 Develop a data definition for representing a list of shots. ∎

Exercise 2.5.2 Design the function *all-shots-move*, which consumes a list of shots and produces one where each shot has been moved with *shot-move*. ∎

Exercise 2.5.3 Design the function *all-shots-draw*, which consumes a list of shots, draws all of them, and produces true if all drawing actions succeed.

Design the function *all-shots-clear*, which consumes a list of shots, clears all of them, and produces true if all drawing actions succeed. ∎

Exercise 2.5.4 Design the function *hit-by-any-shot?*. It consumes a list of *Shot*s and a *UFO*. It produces true if one of the *Shot*s has hit the *UFO*; it produces false if none of the *Shot*s has hit the *UFO*. ∎

Exercise 2.5.5 Design the function *shots-erase-and-clear*. It consumes two lists of shots. The first one is the list of shots that needs to be erased, the second one is the list of shots that need to be drawn. The function produces the second list of shots. See 8. ∎

```
;; UFO AUP → true
;; set up world with UFO and AUP and add event handlers
(define (game ufo aup) ...)

;; KeyEvent World → World
;; create world with moved AUP
(define (new-aup-scene ke w)
  (cond
    [(char? ke) w]
    [(symbol=? 'up ke)
     (make-world (world-ufo w)
                 (world-aup w)
                 (cons (shot-create (world-aup w)) (world-shots w)))]
    [else
     (make-world
       (world-ufo w)
       (erase-and-draw (world-aup w) (aup-move ke (world-aup w)))
       (world-shots w))]))

;; AUP AUP → AUP
;; erase old AUP, draw new one, and produce it
(define (erase-and-draw aup-old aup-new) ...)

;; World → World
;; move ufo and create world with it
(define (new-scene w)
  (cond
    [(hit-by-any-shot? (world-shots w) (world-ufo w))
     (announcement (end-of-time))]
    [else
     (make-world
       (ufo-erase-and-draw (world-ufo w) (ufo-move (world-ufo w)))
       (world-aup w)
       (shots-erase-and-draw
         (world-shots w) (all-shots-move (world-shots w))))]))

;; UFO UFO → UFO
;; erase old ufo and draw new one, unless it's at the bottom
(define (ufo-erase-and-draw old-ufo new-ufo) ...)
```

Figure 8: Crashing or shooting down a UFO with many shots

Exercise 2.5.6 Revise *announcement* exercise 2.4.7. Instead of a *Shot/f*, the function now consumes a list of *Shot*s. ∎

It's time to play. And you're ready to play. The program fragment in figure 8 contains those pieces of code from figure 7 that require changes due to our revised definition of *World*. When combined with previous code and the solutions to the exercises in this section, the program allows AUPs to fire many shots at the UFO. After you have played enough, try to understand and write down how the program evolved and how we planned out this series of exercises. Then think of a game that you want to implement and develop a plan for your game.

2.6 Eliminating Similarities

2.6.1 Abstracting Similar Functions

PREREQUISITE: 19 Similarities in Definitions
Take a second look at figure 1. The program contains two almost identical auxiliary functions: *ball-paint* and *ball-erase*:

```
;; Ball → true                          ;; Ball → true
;; paint the ball from the canvas       ;; earse the ball from the canvas
(define (ball-paint x-ball)             (define (ball-erase x-ball)
  (draw-solid-disk                        (clear-solid-disk
    (make-posn x-ball 50) 10 'red))         (make-posn x-ball 50) 10 'red))
```

Now suppose we want the same program to move a stick figure, not a plain ball. In that case, the program would have to draw and clear the disk and a thin rectangle below the disk. That is, we would need to write these two functions:

```
;; Posn → true                          ;; Posn → true
(define (shape-paint a-posn)            (define (shape-erase a-posn)
  (and                                    (and
    (draw-solid-rect a-posn 1 10 'red)      (clear-solid-rect a-posn 1 10 'red)
    (draw-solid-disk a-posn 3 'red)))       (clear-solid-disk a-posn 3 'red)))
```

Clearly, the two pairs of functions are prime examples of functions with similar definitions. We can easily abstract over both pairs. For the second pair, we get this general function:

```
;; (Posn Number Number Symbol → true)
;;    (Posn Number Symbol → true) Posn → true
(define (graphics-shape solid-rect solid-disk a-posn)
  (and (solid-rect a-posn 1 10 'red)
       (solid-disk a-posn 3 'red)))
```

To get back the two functions, we just pass in the appropriate primitives:

```
;; Posn → true
(define (shape-paint p)
  (graphics-shape
    draw-solid-rect draw-solid-disk p))
```

```
;; Posn → true
(define (shape-erase p)
  (graphics-shape
    clear-solid-rect clear-solid-disk p))
```

The advantage is that, in many cases, we can now change just a single function to get an entirely new shape to move across a canvas.

Exercises

Exercise 2.6.1 Replace the functions *ball-paint* and *ball-erase* in figure 1 with *shape-paint* and *shape-erase*. Don't forget to add *graphics-shape* so that the functions actually work. Then modify *graphics-shape* so that the program moves

1. a square of size 3;
2. a cross-hair that intersect at the given *Posn*;
3. a pair of intersecting disks (each of size 5) that contain the given *Posn* in their intersection. ∎

Exercise 2.6.2 Develop an abstract function for *ufo-draw* and *ufo-clear* from exercise 2.1.4. ∎

Exercise 2.6.3 Develop an abstract function for *aup-draw* and *aup-clear* from exercise 2.2.4. ∎

Exercise 2.6.4 Develop an abstraction for *shot-draw* and *shot-clear* from exercise 2.4.4. ∎

2.6.2 Using Loops

PREREQUISITE: 21.2 Finger Exercises with Abstract List Functions
In addition to abstracting over similar functions, it is also good practice to define functions with applications of existing abstractions. Scheme provides a number of "loops", i.e., functions that traverse a piece of data and apply some give function to each "stop" during the traversal. For example,

```
(map shot-move
     (list (make-posn 100 500)
           (make-posn 100 460)
           (make-posn 120 420)))
```

applies the function *shot-move* to each *Shot* (i.e., *Posn*) on the given list. The result is the list

```
(list (make-posn 100 492) (make-posn 100 452) (make-posn 120 412))
```

In short, the expression moves an entire list of shots.

Exercises

Exercise 2.6.5 Use map to define *all-shots-move* from exercise 2.5.2. ∎

Exercise 2.6.6 Use andmap to define *all-shots-draw* and *all-shots-clear* from exercise 2.5.3. ∎

Exercise 2.6.7 Use ormap to define *hit-by-any-shot?* from exercise 2.5.4. ∎

Now that play some more, but use this second draft of the program.

3 Suggestions

3.1 Worm

The Worm game is one the oldest computer games around. The world of Worm consists of a square landscape with a worm and a piece of food. From the beginning, the worm travels across the landscape in some direction. The player can change the worm's direction via keystrokes using the four arrow keys. The worm continues to travel in the direction of the last command. When the worm gets close enough to the food, it eats the food

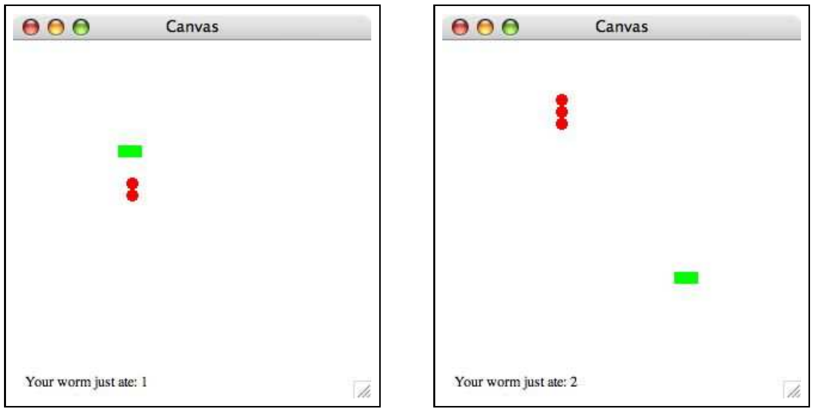

Figure 9: The Worm Game

and grows by one segment. Once the food is eaten, new food shows up somewhere else on the screen.

The goal of the game is to grow the worm as large as possible. Two obstacles prevent the worm from growing forever. First, if the worm runs into itself, it is said to eat itself, which kills it and the game is over. Second, the worm's head can run into one of the four surrounding walls, which has the same consequence. In other words, the goal of the game is to grow the worm as large as possible before it runs into the wall or eats itself.

3.2 BlockScape

BlockScape is a variation on the Tetris theme, which is also a computer game from the early ages of personal computers. The BlockScape world contains on flying block (a rectangle of a certain size) and a list of blocks that have accumulated on the ground. Every time a block lands on the floor or on one of the blocks lying there, a new block appears at the top of the picture. The player can navigate the flying block with the left and right arrow key, to which the block responds with appropriate movements.

The goal of the game is to land as many blocks as possible before the tallest accumulation of blocks reaches the black line (see figure 10). Although this appears trivial at first, the trick is that this game is played a very high speed so that the blocks descend very quickly. On a reasonably modern computer the clock (see *big-bang*) should tick every .01 seconds.

Note: Navigating the blocks requires tests that determine whether a

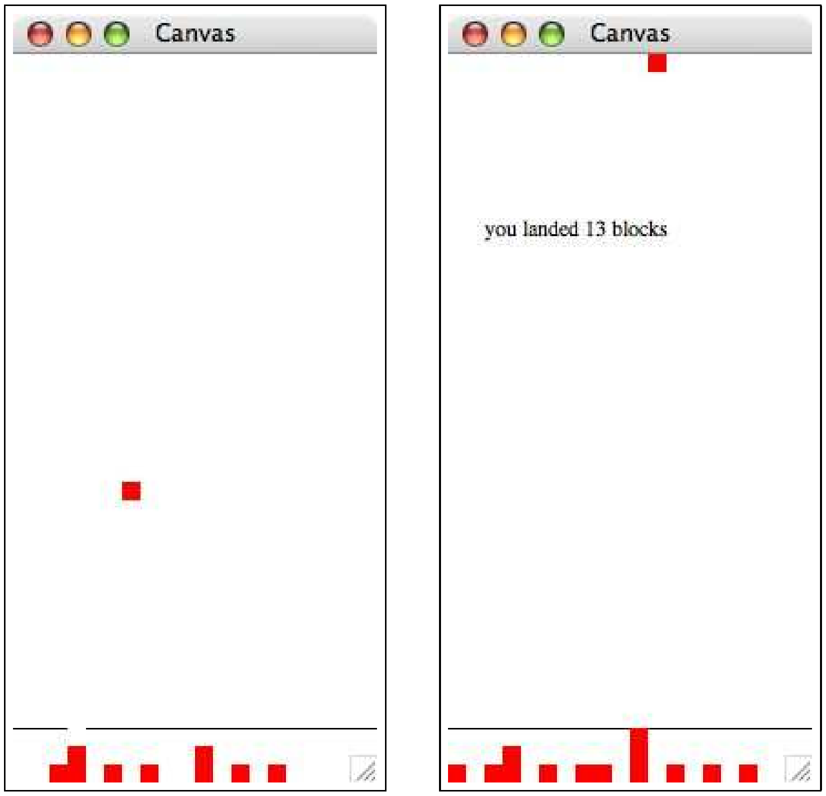

Figure 10: The Blocks Game

block that moves left or right would run into (or through) a block that is already on the ground or whether it would leave the entire picture, which it can't. Landing the block requires a similar test. Specifically, the program must know whether one block is on top of each other.

At first glance, these tests mean that blocks should only move in certain discrete steps, that the picture frame has a certain size, and so on. Instead of relying on the nature of exact rational numbers in Scheme, we recommend that you instead use approximate test. For example, for the "on top of one another" test, you may wish to ask whether the x coordinates of the two blocks are approximately the same and whether the lower-left corner of the top block is approximately near the upper-right of the bottom block.