Extended Exercise: Animations (Draft 1)

A Supplement to "How to Design Programs"

1 Numbers, Expressions, and Simple Programs

Programming is all about representing the world and calculating with these representation. Consider a game program that allows players to shoot down UFOs. The program needs to represent the UFO and the shots and many more things so that it can keep track of them and so that it can draw them on the computer screen. Representing the UFO means to use numbers, combination of numbers, and other forms of data that we don't know yet. The program then computes new data from these given pieces of data and it may use them to draw something, too. In this section, we learn to deal with the basics of representing real or imagined objects in Scheme as numbers, how to calculate with them, and how to draw them so that they appear to move.

1.1 Flying a UFO

Say you want to write a game that involves shooting UFOs. One of the things you have to figure out first is how the UFO descends on a computer window, also known as a *canvas*.

Figure 1 depicts a descending UFO, drawn as a "flying saucer", on a canvas and on a mathematical grid.[1] Let's see how we can simulate this kind of movement with a program. We begin with some mathematics but we are going to see moving pictures real soon. In Algebra, your teacher asked you to create tables like this:

$t =$	0	1	2	3	4	...	7
$X =$	20	30	40	50	60	...	?

The table tells us what the X coordinate of the UFO is when t is 0, 1, 2, and so on.

[1]Unlike in mathematics, the vertical grid numbers grow downwards. That's just one of those differences between mathematics and programming that you need to accept.

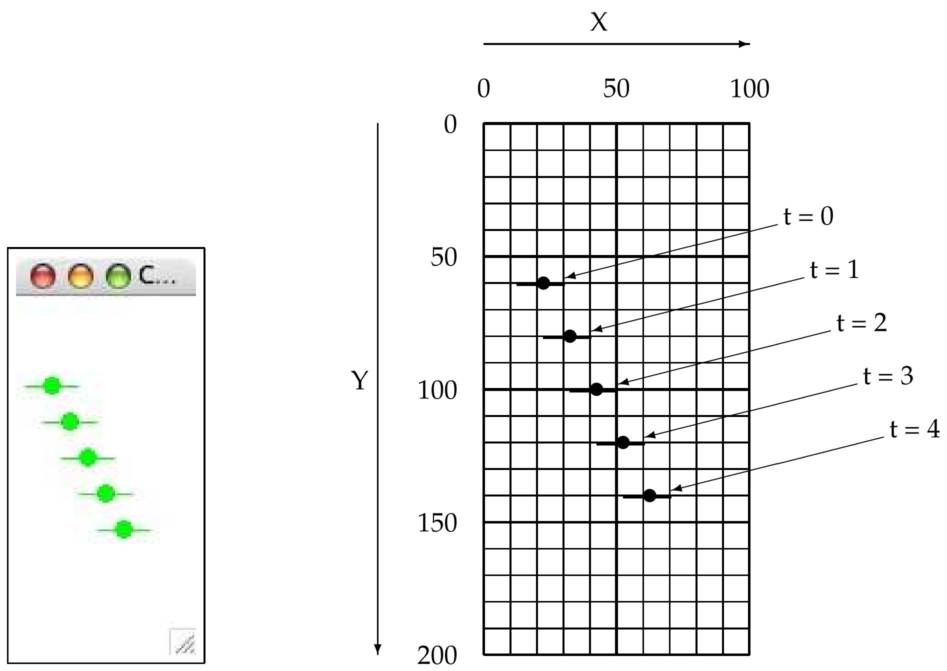


Figure 1: A UFO descending to the ground

Creating this kind of table helps people figure out where the UFO is at all times. For example, it's relatively easy to see that when t is 5, the UFO is at 70. Where is it when t is 7 or when t is 9? Using a table is inconvenient, however, because for something like a computer game we may need to know the X coordinate of the UFO for dozens, hundreds, and thousands of values of t. For this purpose, it is better to figure out an *algebraic formula* that describes what X is for a given value of t.

Here is the algebraic formula that relates X to t:

$$X = 10 \cdot t + 20$$

Right now, it just showed up. Usually your teacher asks you to come up with such a formula on your own, generalizing from a table like the one above. In that case, it is good practice to test whether it works properly. *Testing* means to pick some values for t for which we know what X should be and to check that the formula really produces the correct value of X.

When *t* is 0, the formula becomes

$$X = 10 \cdot 0 + 20$$

and with a little bit of calculation we see that

$$X = 20$$

as expected from the figure and the table. The calculation for the case when *t* is 1 is equally simple:

$$X = 10 \cdot 1 + 20$$

which we can simplify to $10 + 20$ and therefore

$$X = 30$$

also as predicted in the figure and the table.

Exercises

Exercise 1.1.1 Fill in the missing numbers in this table:

$t =$	0	1	2	3	4	…	9
$Y =$						…	

The table describes how far down the UFO is at the various times.

Formulate an algebraic formula that describes the relationship between *t* and Y. Test the algebraic formula with the examples from your table. ∎

Exercise 1.1.2 Take a look at figure 2. It depicts a shot that was fired from the ground going straight up. The left version is again a canvas, and the right version is a similar picture drawn on a grid.

Fill in the following table, which relates the Y position of the shot with the time *t*:

$t =$	0	1	2	3	4	…	6
$Y =$						…	

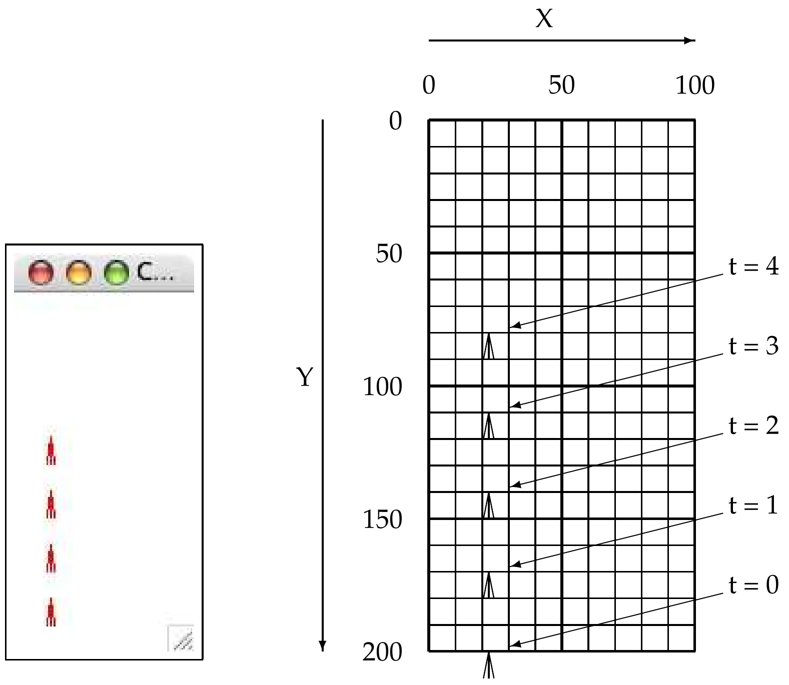


Figure 2: A shot from the ground

At what *X* position is the shot all the time?

Formulate an algebraic formula that describes the relationship between *t* and *Y* for the shot. Test the algebraic formula with the examples from your table. If you find a mistake, does it have to be in the formula? Could it be in the table? Or both? Also formulate an algebraic formula that specifies what *X* is given the current value of *t*. ∎

It's time to start writing our game program. First, we tackle the UFO's flying action. Specifically, we want a program that computes the UFO's *X* coordinate given the time *t*:

```
(define (X t) (+ (* 10 t) 20))
```

If we squint, the program looks just like the algebraic formula:

$$X = 10 \cdot t + 20$$

Let's look at the differences:

1. The most obvious difference is that the program is littered with parentheses. Compare the variable expression on the left of the equality

   $$10 \cdot t + 20$$

   with the Scheme expression in the program:

   ```
   (+ (* 10 t) 20)
   ```

   Clearly, every Scheme expression is surrounded with (and), and it always mentions the operator first. Hence, there is never a question what is to be done next and in what order.

2. The word **define** is a big marker. It says that we are defining a *function*, which is the real name for such algebraic formulas. Scheme programs consist of many functions, and that's all there is to programming mostly.

3. The middle piece tells us two things: the name of the expression, *X* in this case, and the name of the variable, *t*, on which it depends.

It looks different from regular algebra, but what differs is the looks, not the meaning.[2]

Now that we have defined the function in Scheme, we must learn to use it. As we just discussed, the definition tells us that *X* depends on *t*. This means that when we want to use *X*, we must supply a value for *t*, and then the Scheme program replies with the value of *X*. Here are some example:

```
> (X 0)
20
> (X 1)
30
> (X 2)
40
> (X 3)
50
```

In other words, to supply a value to a function, we write "(", the name of the function, a value for the independent variable of the function (*t* here), and a ")". What we get is an expression just like (+ 1 1) or (* 2 2). DrScheme, which is basically a calculator for Scheme functions, then

[2]As you will see over time, this style has advantages, and it takes little time to get used to it.

evaluates this expression for us, using the rules from algebra that we know so well. Indeed, if you use DrScheme's Stepper, it evaluates the expression exactly like you evaluate an algebraic expression for your homework.

If you compare these four evaluations with the table above, you see that the results are just the ones we expected: 20, when t is 0; 30 when t is 1; and so on. Of course, we can also try randomly chosen inputs such as 8 or 22, just to see what X is for such values of t:

```
> (X 8)
100
> (X 22)
240
```

The game program uses this function all the time to figure out where to draw the UFO.

Exercises

Exercise 1.1.3 In exercise 1.1.1 you figured out that the algebraic formula for Y given t:

$$Y = \cdots t \cdots$$

Use this formula to develop the function Y, which—like X—consumes t and computes the Y coordinate for the UFO. Test the function with inputs from the table that you used to develop the formula. Compare the results with the expected outputs in the table. ∎

Exercise 1.1.4 Exercise 1.1.2 asked you to figure out how a shot moves across a grid. That included making up tables and algebraic formulas for the X and Y coordinates formula. Remember that the formulas describe how to compute the coordinates given a value for t.

Use these formulas to develop the function *shot-X* and *shot-Y*, which consume t and compute the X and Y coordinates for the shot. Test the functions with inputs from figure 2. Compare the results with the expected outputs from the table. ∎

1.2 Drawing a UFO

Now we know how to compute where a UFO is, but we don't know yet how to draw one let alone create a canvas in which we can draw one. This is where "libraries" come in. A library—here also called TEACHPACK—is a collection of functions (or instructions) that assist us with just such tasks.

For drawing, we use the **simple-draw.ss** teachpack. It provides four functions that matter now:

1. (*start width height*) creates a canvas of *width* $\times$ *height* pixels;
2. (*draw-solid-disk0 x y r c*), which draws a colored disk at position (*x*,*y*), with radius *r* and color *c*;
3. (*draw-solid-rect0 x y w h c*), which draws a colored rectangle at position (*x*,*y*), of *w* $\times$ *h* pixels, and color *c*;
4. (*draw-solid-line0 x0 y0 x1 y1 c*), which draws a colored line of color *c* between the two positions (*x0*,*y0*) and (*x1*,*y1*).

That's all there is to **simple-draw.ss**.

Except that we don't know what to write where these functions expect a color. Until now, all of our functions have consumed numbers and produced numbers. It is therefore almost natural to expect some encoding that says 0 represents white, 1 represents yellow, 2 represents orange, and so on. Of course, after a while we wouldn't remember which number represents which color, so this is clearly not a good idea. The simple drawing teachpack therefore uses a more intuitive way to say we want a blue line or a red disk:

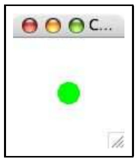
```
'white    'yellow    'red    'blue    'green    'black
```

In short, think of a color word and put a tick in front of it, but remember that this library doesn't know all the colors.

Let's experiment with these functions a bit:[3]

```
> (start 100 200)
true
```

This creates a canvas on your computer's monitor that is 100 pixels wide and 200 pixels high. To draw a green disk at the center of this canvas, we enter this instruction:

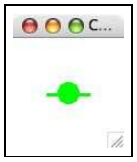


```
> (draw-solid-disk0 50 50 10 'green)
true
```

The resulting disk has a radius of 10 pixels, as the accompanying picture shows. If we also draw a thin rectangle 20 pixels to the left of the disk's center:

[3]Ignore the "true" part for now.

```
> (draw-solid-rect0 30 50 40 3 'green)
true
```

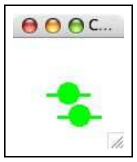


the figure looks approximately like a flying saucer. To get a better understanding of this picture, figure out why the width of the rectangle is 40.

A descending UFO isn't drawn once, but many times. Otherwise it looks stationary. More precisely, the game program must draw the UFO when t is 0, 1, 2, and so on. So let's figure out how we can accomplish this step by step.

To draw the UFO one more time, we need to copy the above two lines and change the *X* and *Y* coordinates:

```
> (draw-solid-disk0 60 70 10 'green)
true
> (draw-solid-rect0 40 70 40 3 'green)
true
```



Recall that for each step, the UFO moves down by 20 pixels and right by 10. This leaves us with two UFOs now.

Clearly, copying and editing code is not feasible if we want to draw the UFOs hundreds of times. Instead, we define a function for drawing UFOs and use it as often as needed. Defining this function is similar to defining a function that computes the *Y* coordinate of the UFO. Specifically, we try to reuse the expressions (or instructions) that we used to draw the UFO twice and we generalize from it:

show table with expressions?

```
(define (ufo-draw ...)
  (and (draw-solid-disk0 ... ... 10 'green)
       (draw-solid-rect0 ... ... 40 3 'green)))
```

Since we drew a disk and a rectangle to draw the UFO, the function says that to draw a UFO first draw a disk **and** then a rectangle. Of course, the dots tell us that the function is still incomplete.

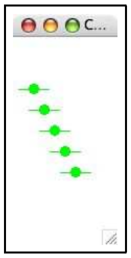
The dots signal that we don't know yet what to write down at these places. More precisely, the dots in the draw-solid- instructions say we don't know the *X* and *Y* coordinates of the UFO. We know, however, that the location of UFO depends on the value of *t* and that the functions *X* (from above) and the function *Y* (from exercise 1.1.3) describe how to get the respective coordinates when we have *t*. This suggests the following refinement of the definition:

```
(define (ufo-draw t)
  (and (draw-solid-disk0 (X t) (Y t) 10 'green)
       (draw-solid-rect0 (X t) (Y t) 40 3 'green)))
```

In other words, the function *ufo-draw* consumes *t*, computes the *X* and *Y* coordinates of the UFO for *t*, and then draws a disk and a rectangle.

Using *ufo-draw* it is easy to draw the UFO four times or a hundred times:

```
> (draw-ufo 0)
true
> (draw-ufo 1)
true
> (draw-ufo 2)
true
> (draw-ufo 3)
true
> (draw-ufo 4)
true
```



Recall that for each step, the UFO moves down by 20 pixels and right by 10. In the end, we have with five UFOs.

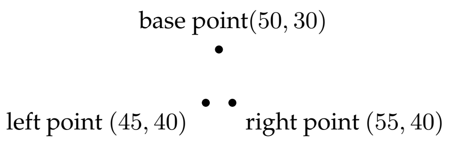
Exercises

Exercise 1.2.1 Create a canvas of 100 by 200 pixels and draw a shot close to the bottom of the canvas. For now, use thin, tall rectangles to depict shots. Draw a second shot.

Define the function *shot-draw*, which—like *ufo-draw*—consumes *t* and draws a shot. It computes the *X* and *Y* coordinates of the shot using the functions from exercise 1.1.4. ∎

Exercise 1.2.2 Shots don't look good as squares. Let's use thin, upward pointing, equilateral triangles instead.

To start with, create a canvas of 100 by 100 pixels, and draw a small equilateral triangle into it with these base points:



Connect the base points with three lines. What is the height of the triangle? What is the width of its base?

Next, draw a triangle of the same width and height but with the highest point at $(70, 10)$.

Draw two more shots, one at $(20, 30)$ and another one at $(80, 30)$. You can think of these last two and the first shot as one and the same shot moving upwards.

Use your experience with the four triangle to define a function that can draw a triangle of this size, given the coordinates of the base point:

```
;; given: xtop is a the x coordinate of the top-most point,
;; ytop is the y coordinate
(define (triangle-draw xtop ytop)
  (and ...
       ...
       ...))
```

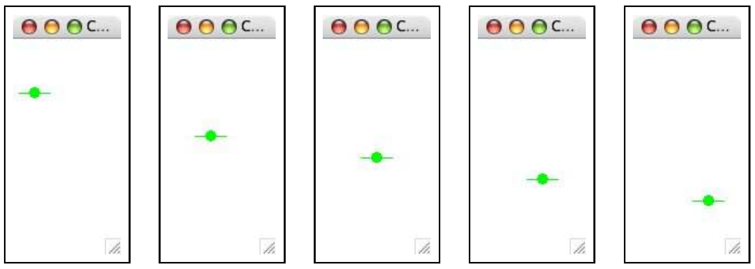
The first two lines are called comments, and they remind you what *xtop* and *ytop* are, when you will read this program in two months or three years from now.

Finally, define the function *shot-draw*, which draws a shot as a triangle. It consumes a single number, the time *t*, and draws a triangle on a canvas of size 100 by 200:

```
;; given: t
(define (shot-draw t)
  ...)
```

Use the functions *shot-X* and *shot-Y* from exercise to determine where the base point is. Hint: You should use *triangle-draw*. ∎

So far we know how to draw the UFO when *t* is 0, 1, 2, 3 and 4, but this doesn't really show how the UFO moves. It just shows the same UFO at five different times. In a computer game, we would just see the UFO glide across the screen like this:



The left-most screenshot depicts the scene when *t* is 0, and the right one represents the scene when *t* is 4.

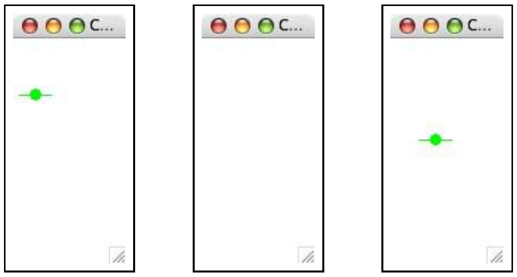
Put differently, as the program updates the scene, going from *t* to $t + 1$, it must erase the old UFO and draw the new one. So what does erasing mean? Say the UFO is green and the canvas is white. If the program draws a UFO in *white* at the *same place*, the UFO is no longer visible. So erasing really just means "draw the shape in white" and is nothing special.

Since the program must erase the UFO as often as it draws it, let's define a function for this action. The only difference between this erasing function and the drawing is the color that we give the drawing instructions. Hence, copying the definition of *ufo-draw* and replacing 'green with 'white should work:

```
(define (ufo-erase t)
  (and (draw-solid-disk0 (X t) (Y t) 10 'white)
       (draw-solid-rect0 (X t) (Y t) 40 3 'white)))
```

Here is a test:

```
> (ufo-draw 0)
true
> (ufo-erase 0)
true
> (ufo-draw 1)
true
```



The first instruction draws the UFO when *t* is 0 and produces the left-most canvas in the sequence. The second instruction erases the UFO and produces the blank canvas in the middle. The last instruction draws the UFO when *t* is 1. As we can see from the sequence of screen shots, our pair of functions performs as expected, and if we could somehow repeat these kind of instructions quickly, say, once a second or even faster, we would have the beginning of a game program.

Still, this is *not* a good way to write this function, especially if the program grows and consists of hundreds or thousands of such small functions. To understand why this is bad, imagine you want a blue background. It's easy to accomplish; just have your function draw a blue rectangle of the size of the canvas before anything else. What would you have to change in the definition of *ufo-erase* to get the moving effect for the UFO? Since there are two occurrences of 'white, a good and valid guess is that we must replace both occurrences with 'blue.

Imagine a second change. Say the UFO should have an antenna. For

```
;; given: t, the time, and c, a color
(define (ufo-paint t c)
  (and (draw-solid-disk0 (X t) (Y t) 10 c)
       (draw-solid-rect0 (X t) (Y t) 40 3 c)))

;; given: t, the time
(define (ufo-draw t) (ufo-paint t 'green))

;; given: t, the time
(define (ufo-erase t) (ufo-paint t 'white))
```

Figure 3: Drawing and erasing UFOs properly

simplicity, the function can draw the antenna as a short line that sticks out of the disk, pointing upwards. Of course, adding this line drawing instruction to *ufo-draw* is not enough. In addition, we need to add it to *ufo-erase* and replace 'green with the background color.

Because we copied the code for *ufo-draw* and created *ufo-erase* from it, changes to the program require duplicate work. Although such duplicate work looks harmless right now, especially with such small functions as ours, it is a good idea to avoid it from the very beginning.

The trick to avoiding duplicate work is *abstraction*, a fancy word for defining a function. In our case, we want to define a function that is like *ufo-draw* and *ufo-erase* but it can do both. Since the difference between the two functions is the color that they use, the natural way to merge them into one function is to use an argument for the color. That is, we define a function that takes two arguments: the value of *t* and a color. It then draws the UFO shape in the given color, using (*X t*) and (*Y t*) for the coordinates.

Take a look at figure 3. The first definition—named *ufo-paint*—is this general function. Take a look at the revised definitions of *ufo-draw* and *ufo-erase* in this figure. They just use *ufo-paint* to draw the UFO at the proper place and in the proper color. If we now change the shape of the UFO, all we have to do is change *ufo-paint*, once! If we want to change the background color to blue, we can do so, by changing *ufo-erase*, in one place. In short, by working a bit extra now, we can save ourselves and other people working on the function a lot of time in the future.

1.3 Animating the UFO's Flight

In the first subsection, we learned to compute the X and Y coordinates from the value of t. In the second subsection, we learned to draw a UFO at its proper coordinates in various colors. And we saw how to move the UFO across a canvas. Unfortunately, the UFO's movement is too slow for a game. Worse, thus far, we merely know how to move the UFO manually via instructions issued at the prompt of DrScheme's Interactions window.

What we really want is a mechanism for redrawing the canvas every so often automatically. To support just this kind of action, the **simple-draw.ss** teachpack supports three more instructions:

1. (*big-bang n x*) starts the clock and makes it tick every *n* seconds; it creates a world from *x*;

2. (*on-tick-event tock*) sets the tick handler to *tock*, which is applied to the current world at every tick of the clock and produces the next world;

Clearly, to understand these functions and to see what they do, we must understand what it means to create a world and to compute a new one.

A world in the sense of the teachpack is something that we want to track during our computations. Usually it is an entire collection, say a UFO and a bunch of shots and perhaps some surface vehicles that fire these shots and so on. Since all we know, however, are numbers, our world is just a number. For example, if all we want to do is move a single UFO across the canvas, we can just use *t* to represent the world. From that, other functions can compute the UFO's coordinates and a third function can draw the UFO at these coordinates.

This suggests the following to get started:

```
> (start 100 200)
true
> (big-bang .5 0)
true
```

The first instruction creates the canvas. The second starts the clock and sets *t* to 0.

The critical step is to tell the teachpack which function we want to run at every tick of the clock. Let's call this function *next*. Once we enter

```
> (on-tick-event next)
true
```

the teachpack applies *next* to the first world, call it *w0*, and *next* produces another world, say *w1*. Then the teachpack applies *next* to *w1* and obtains *w2*, and so on. Put differently, the teachpack uses *next* in a chain that looks like this:

```
...
 (next
  (next
   (next
    (next
     (next w0))))) ...
```

Every use of *next* is synchronized with the tick of the clock.

In our specific example, the world is *t* and we know that *t* starts with 0, and then becomes 1, 2, and so on. Hence, the function that consumes *t* and produces the next one just adds 1 to *t*:

```
;; draft 1
(define (next-time t) (+ t 1))
```

Given that *w0* in our example is 0, we now know that the *teachpack* executes something like this:

```
...
 (next-time
  (next-time
   (next-time
    (next-time
     (next-time 0))))) ...
```

Of course, this accomplishes nothing yet. The first use of *next-time* consumes 0 and produces 1, the second consumes the 1 and produces 2, and so forth, but no UFO is drawn or erased.

To get out "world transforming" functions to draw something on the canvas, we need one more piece of knowledge about **simple-draw.ss**. Take a look at this function definition:

```
;; draft 2:
(define (next-time t)
  (draw (ufo-erase t)
        (ufo-draw (+ t 1))
        produce (+ t 1)))
```

Like the first draft, this version of *next-time* consumes the value of *t* and, as the bold **produce** says, produces (+ *t* 1). In addition, however, the function also performs two **draw**ing tasks: it erases the UFO at *t* and draws it at (+ *t* 1).

```
;; given: t, compute the x coordinate for the UFO
(define (X t) (+ 20 (* 10 t)))

;; given: t, compute the y coordinate for the UFO
(define (Y t) (+ 50 (* 20 t)))

;; given: t and a color c,
;; paint the UFO at the coordinates for t in color c
(define (ufo-paint t color)
  (and (draw-solid-disk (make-posn (X t) (Y t)) 5 color)
       (draw-solid-rect (make-posn (- (X t) 15) (Y t)) 30 1 color)))

;; given: t, paint the UFO at the coordinates for t in 'green
(define (ufo-draw t) (ufo-paint t 'green))

;; given: t, erase the UFO
;; by painting it at the coordinates for t in the background color
(define (ufo-erase t) (ufo-paint t 'white))

;; given: t, erase the UFO at t, draw it
;; at (+ t 1, and return (+ t 1) as the new value
(define (tock t)
  (draw (ufo-erase t)
        (ufo-draw (+ t 1))
        produce
        (+ t 1)))
```

```
;; create the canvas
(start 100 200)

;; start the clock, make it tick every .5 seconds; the initial "world" is 0
(big-bang .5 0)

;; every time the clock tick, apply tock to the current world
;; and let it produce the next world
(on-tick-event tock)
```

Figure 4: A UFO Animation

One good way to study the behavior of *next-time* is to tabulate the uses of the function:

clock tick	.5	1.0	1.5	...
given: *t*	0	1	2	...
draw	(*ufo-erase* 0)			...
	(*ufo-draw* 1)	(*ufo-erase* 1)		...
		(*ufo-draw* 2)	(*ufo-erase* 2)	...
			(*ufo-draw* 3)	...
produce	1	2	3	...

Each column shows when *next-time* is used, the value of *t* that it consumes, the **draw** actions that it performs, and the value that it **produce**s. As explained already, the value that it produces is the value that *next-time* consumes next.

To clarify how the drawing actions interact, the table depicts them in a "stacked" manner. This shows that (*ufo-draw* 1) when *t* is 0 is followed by (*ufo-erase* 1) when *t* is 1. Similarly, (*ufo-draw* 2) is followed by (*ufo-erase* 2) when *t* is 2. More concisely, *next-time* ensures that each use of *ufo-draw* is followed by an equivalent use of *ufo-erase*, which erases the UFO before it is redrawn at a different spot.

After all this preliminary work, we know enough to discuss and understand the program in figure 4. The program consists of six function definitions and three expressions, separated by a horizontal line. The bottom half creates a canvas, starts the clock, and tells the teachpack to run the function *tock* on every clock tick. The top half of the figure introduces the two functions for computing a UFO's coordinates, the three functions for drawing and erasing the UFO, and the function *tock* that is run for every tick of the clock. And that's all there is to flying a UFO across your computer's monitor. The next goal is to make the UFO land and to enable the player to shoot at the UFO at least once to prevent it from landing.

Exercises

Exercise 1.3.1 Explore the program in figure 4.

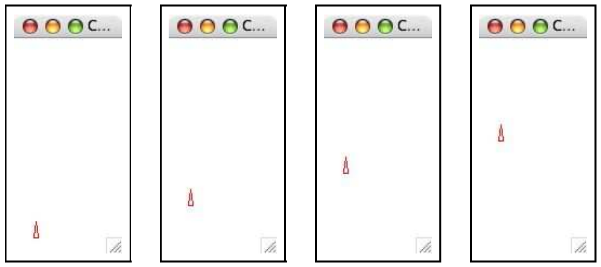
Find out how *tock* in that figure works manually. Enter expression expressions such as (*tock* 0) and (*tock* 1) into DrScheme's Interactions window and see what their results and their effects on the canvas are. Hint: Execute the program in figure 4 but comment out the expression (*on-tick-event tock*).

Make the UFO animation slower and/or faster.

Render the UFO as a yellow shape. Add an antenna. Make the background blue.

Swap the two **draw**ing actions in *tock*. Does this make a difference? Why? Why not? ∎

Exercise 1.3.2 Develop a complete function for simulating the flight of a shot, starting at the bottom of the canvas:



Use the functions from exercises 1.1.4 and 1.2.1. Display the shot as a small disk first and then use the function from exercise 1.2.2 to turn it into a triangle. ∎

Exercise 1.3.3 Combine the UFO and the shot simulation in one canvas. That is, the UFO should descend from the top and the shot should fly upwards from the bottom, starting at the same time on the same canvas. ∎

2 How to Design Programs

2.1 Programs are Function Definitions and Variable Definitions

2.2 Errors

2.3 The Recipe

3 Conditional Expressions and Functions

3.1 Drawing a UFO, Again

3.2 Landing the UFO

3.3 Hitting the UFO with a Shot

3.4 Booleans, Relations, and Conditions

3.5 Functions that Test Conditions

3.6 Designing Conditional Functions

4 Symbolic Information

4.1 Controlling the Anti-UFO Vehicle

4.2 Symbols, Characters, and Strings

4.3 Pictures and Images

5 Compound Data, Part 1: Structures

5.1 Shooting Down UFOs: The First Complete Game