Reactive Animations

- **Todays Topics**
  - **Simple Animations - Review**
  - **Reactive animations**
  - **Vocabulary**
  - **Examples**
  - **Implementation**
    - » **behaviors**
    - » **events**
- **Reading from Hudak's The Haskell School of Expression**
  - **Read Chapter 15 - A Module of Reactive Animations**
  - **Read Chapter 17 – Rendering Reactive Animations**

Review: Behavior

- A `Behavior a` can be thought of abstractly as a function from `Time` to `a`.
- In the chapter on functional animation, we animated `Shape's, Region's, and Picture's.`
- For example:

X coord

Y coord

```
dot = (ell 0.2 0.2)
ex1 = paint red (translate (0, time / 2) dot)
```

Try It

```
ex2 = paint blue (translate (sin time,cos time) dot)
```

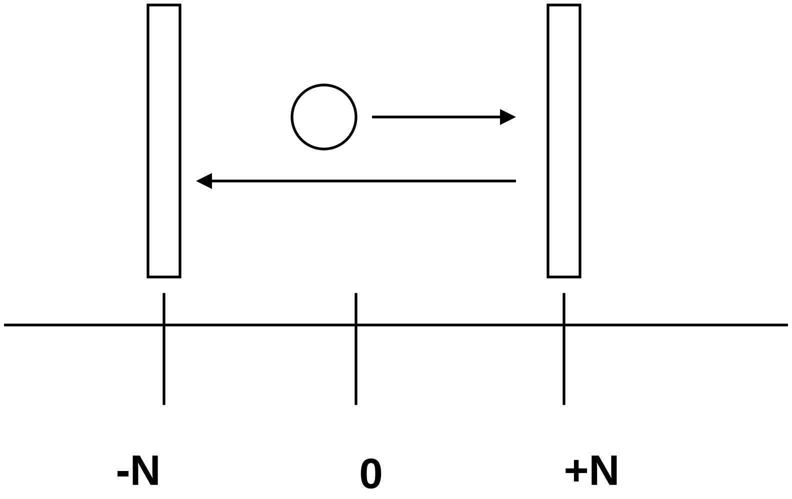
Abstraction

- The power of animations is the ease with which they can be abstracted over, to flexibly create new animations from old.

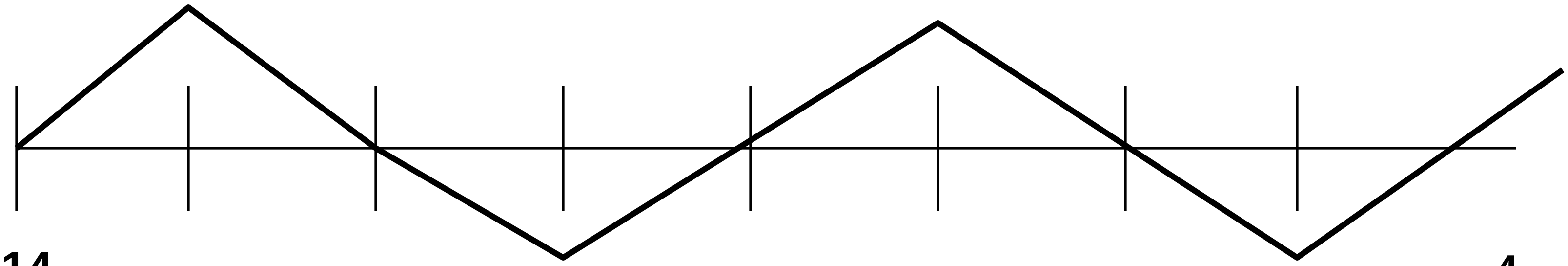
```
wander x y color = paint color (translate (x,y) dot)

ex3 = wander (time /2) (sin time) red
```

Example: The bouncing ball

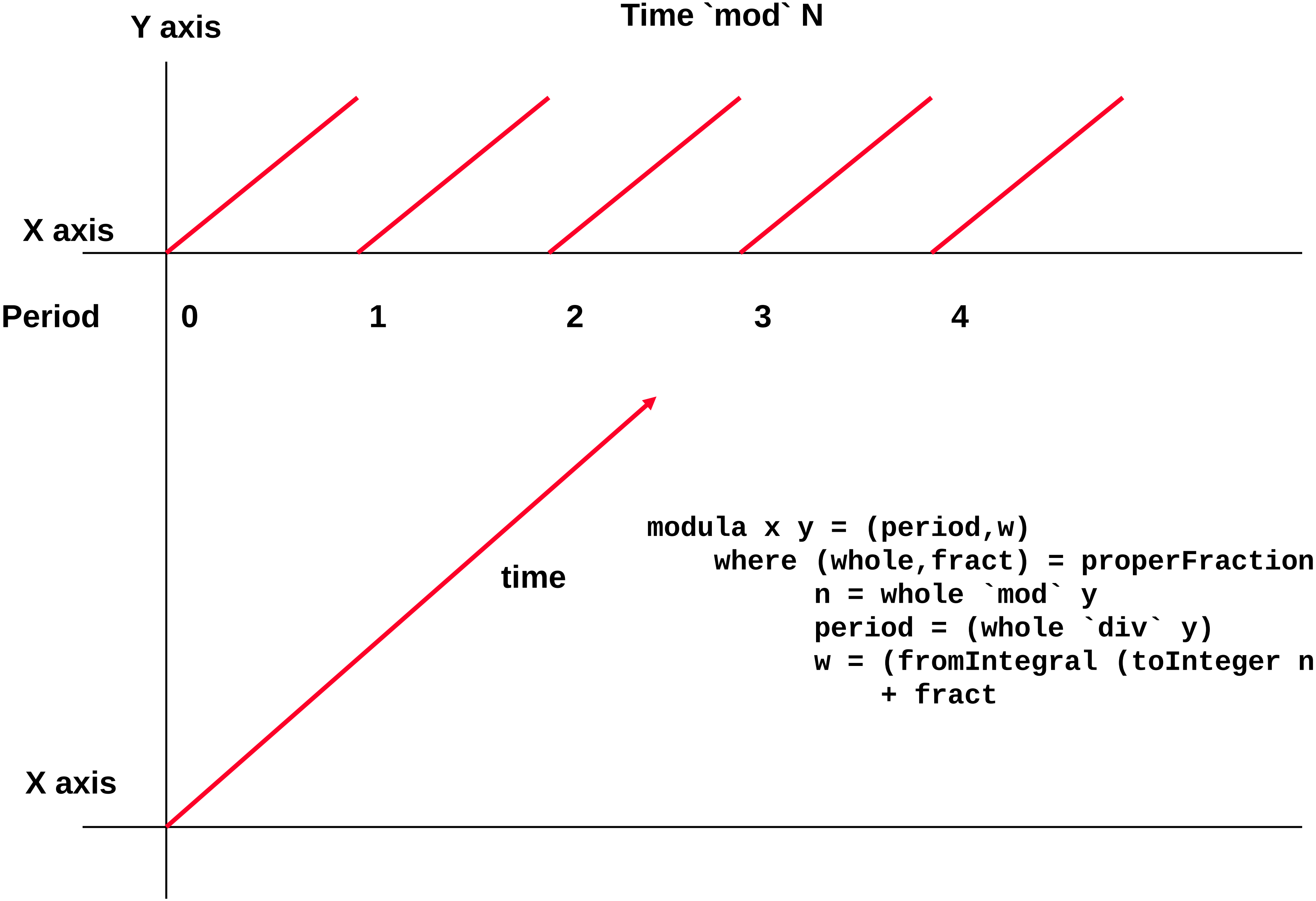
- Suppose we wanted to animate a ball bouncing horizontally from wall to wall



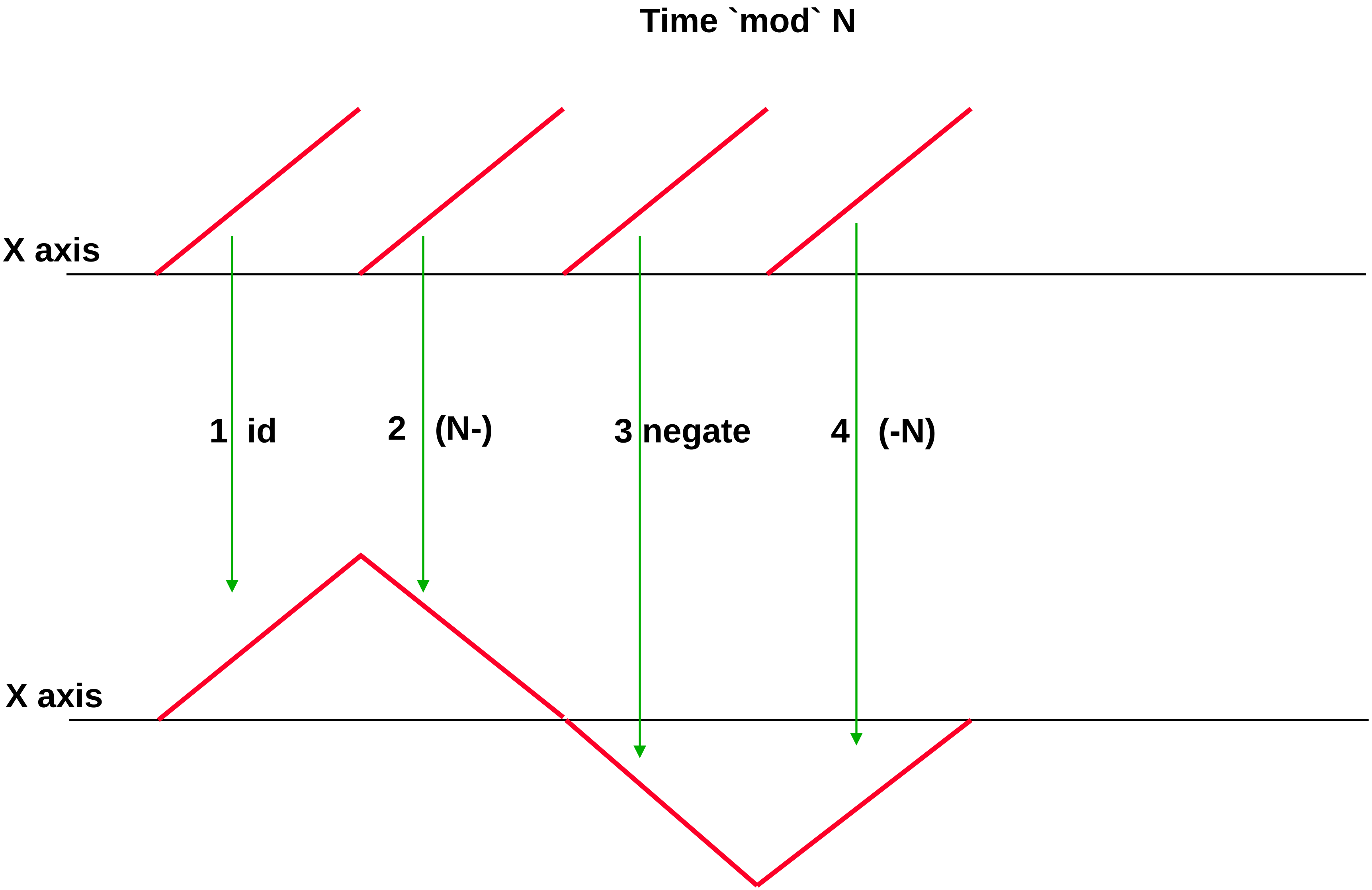
- The Y position is constant, but the x position varies like:



Time `mod` N



```
modula x y = (period,w)
    where (whole,fract) = properFraction x
          n = whole `mod` y
          period = (whole `div` y)
          w = (fromIntegral (toInteger n))
              + fract
```


Time `mod` N
X axis
1 id
2 (N-)
3 negate
4 (-N)
X axis

Implementation

```
bounce t = f fraction
   where (period,fraction) = modula t 2
         f = funs !! (period `mod` 4)
         funs = [id,(2.0 -),negate,(\x -> x - 2.0)]
```

```
ex4 = wander (lift1 bounce time) 0 yellow
```

- **Remember this example. Reactive animations will make this much easier to do.**

Reactive Animations

- With a reactive animation, things do more than just change and move with time according to some algorithm.
- Reactive programs "react" to user stimuli, and real-time events, even virtual events, such as:
  - key press
  - button press
  - hardware interrupts
  - virtual event - program variable takes on some particular value
- We will try and illustrate this first by example, and then only later explain how it is implemented
- Example:

```
color0 = red `switch` (lbp ->> blue)
moon = (translate (sin time,cos time) dot)
ex5 = paint color0 moon
```

A Reactive Vocabulary

- Colors
  - `Red,Blue,Yellow,Green,White :: Color`
  - `red, blue, yellow, green, white :: Behavior Color`

- Shapes and Regions
  - `Shape :: Shape -> Region`
  - `shape :: Behavior Shape -> Behavior Region`

  - `Ellipse,Rectangle :: Float -> Float -> Region`
  - `ell, rec :: Behavior Float -> Behavior Float -> Behavior Region`

  - `Translate :: (Float,Float) -> Region -> Region`
  - `translate :: (Behavior Float, Behavior Float) -> Behavior Region -> Behavior Region`

Operator and Event Vocabulary

- **Numeric and Boolean Operators**
  - `(+), (*) :: Num a => Behavior a -> Behavior a -> Behavior a`
  - `negate :: Num a => Behavior a -> Behavior a`

  - `(>*),(<*),(>=*),(<=*) :: Ord a => Behavior a -> Behavior a -> Behavior Bool`
  - `(&&*),(||*) :: Behavior Bool -> Behavior Bool -> Behavior Bool`
- **Events**
  - `lbp :: Event ()          -- left button press`
  - `rbp :: Event ()          -- right button press`
  - `key :: Event Char        -- key press`
  - `mm  :: Event Vertex      -- mouse motion`

Combinator Vocabulary

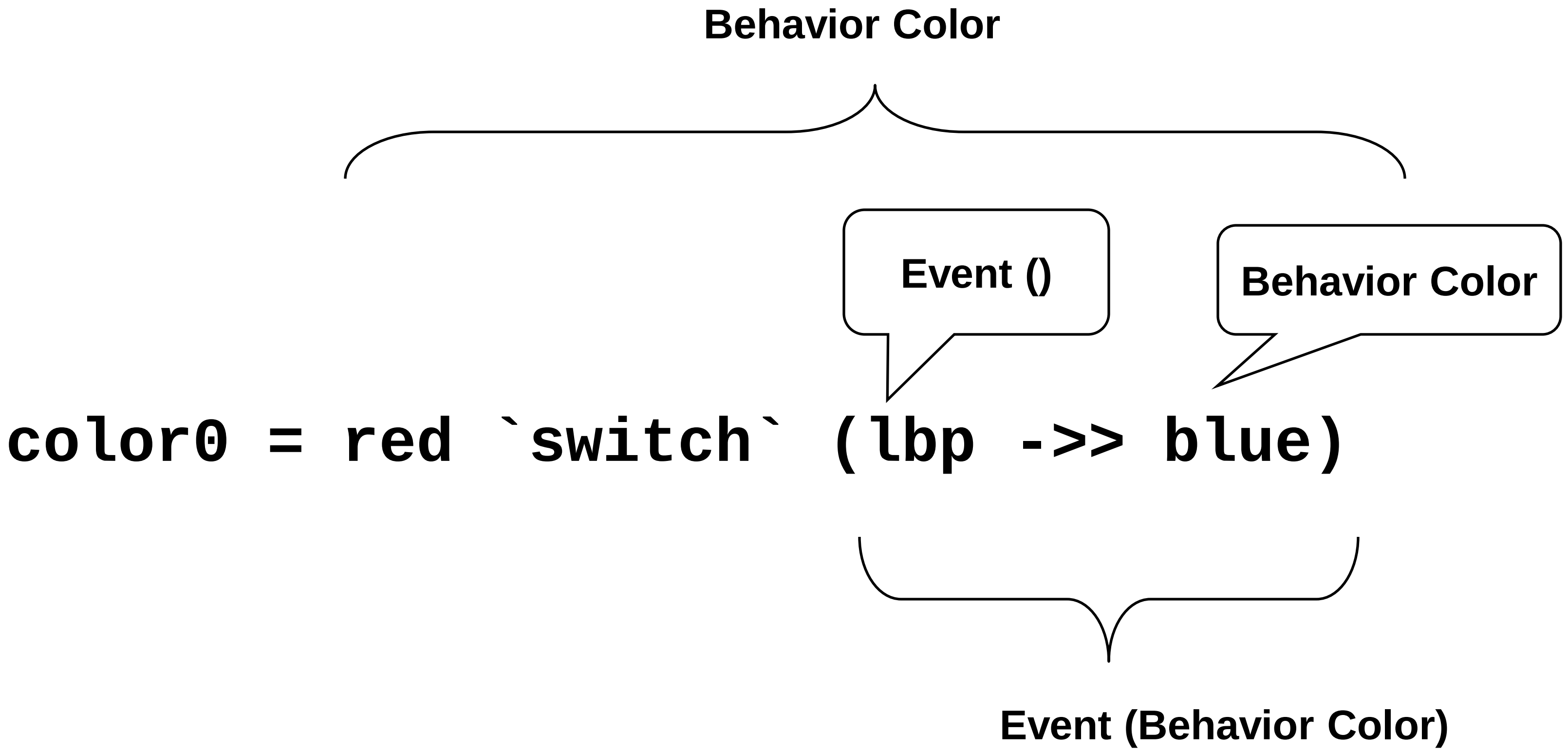
- **Event Combinators**
  - `(->>) :: Event a -> b -> Event b`
  - `(=>>) :: Event a -> (a->b) -> Event b`

  - `(.|.) :: Event a -> Event a -> Event a`
  - `withElem  :: Event a -> [b] -> Event (a,b)`
  - `withElem_ :: Event a -> [b] -> Event b`

- **Behavior and Event Combinators**
  - `switch :: Behavior a -> Event(Behavior a) -> Behavior a`
  - `snapshot_ :: Event a -> Behavior b -> Event b`
  - `step :: a -> Event a -> Behavior a`
  - `stepAccum :: a -> Event(a -> a) -> Behavior a`

Analyse Ex3.

```
red,blue :: Behavior Color
lbp :: Event ()
(->>) :: Event a -> b -> Event b
switch :: Behavior a -> Event(Behavior a) -> Behavior a
```



Behavior Color

Event ()

Behavior Color

```
color0 = red `switch` (lbp ->> blue)
```

Event (Behavior Color)

Either (.|.) and withElem

```
color1 = red `switch`
              (lbp `withElem_` cycle [blue,red])

ex6 = paint color1 moon
```

```
color2 = red `switch`
             ((lbp ->> blue) .|. (key ->> yellow))

ex7 = paint color2 moon
```

Key and Snapshot

```
color3 = white `switch` (key =>> \c ->
           case c of ‘r' -> red
                     ‘b' -> blue
                     ‘y' -> yellow
                     _   -> white  )
ex8 = paint color3 moon


color4 = white `switch` ((key `snapshot` color4) =>>
  \(c,old) ->
           case c of ‘r' -> red
                     ‘b' -> blue
                     ‘y' -> yellow
                     _   -> constB old)
ex9 = paint color4 moon
```

Step :: a -> Event a -> Behavior a

```
size '2' = 0.2  -- size :: Char -> Float
size '3' = 0.4
size '4' = 0.6
size '5' = 0.8
size '6' = 1.0
size '7' = 1.2
size '8' = 1.4
size '9' = 1.6
size _ = 0.1

growCircle :: Char -> Region
growCircle x = Shape(Ellipse (size x) (size x))

ex10 =  paint red (Shape(Ellipse 1 1)
                       `step` (key =>> growCircle))
```

stepAccum :: a -> Event(a -> a) -> Behavior a

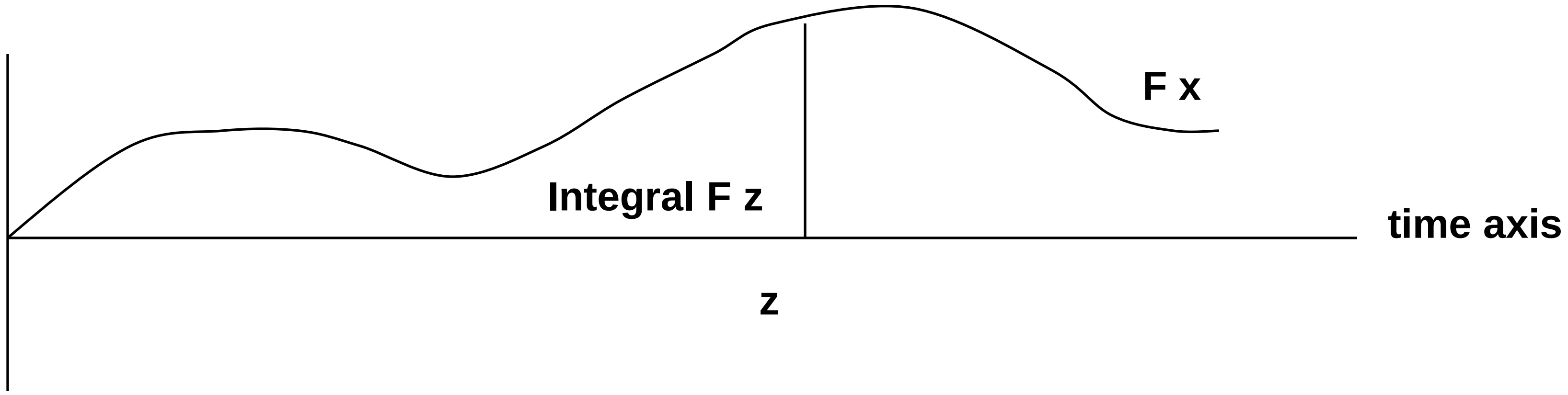
- **stepAccum takes a value and an event of a function. Everytime the event occurs, the function is applied to the old value to get a new value.**

```
power2 :: Event(Float -> Float)
power2 = (lbp ->> \ x -> x*2)      .|.
         (rbp ->> \ x -> x * 0.5)

dynSize = 1.0 `stepAccum` power2
ex11 = paint red (ell dynSize dynSize)
```

Integral

- **The combinator:**
  - `integral :: Behavior Float -> Behavior Float`

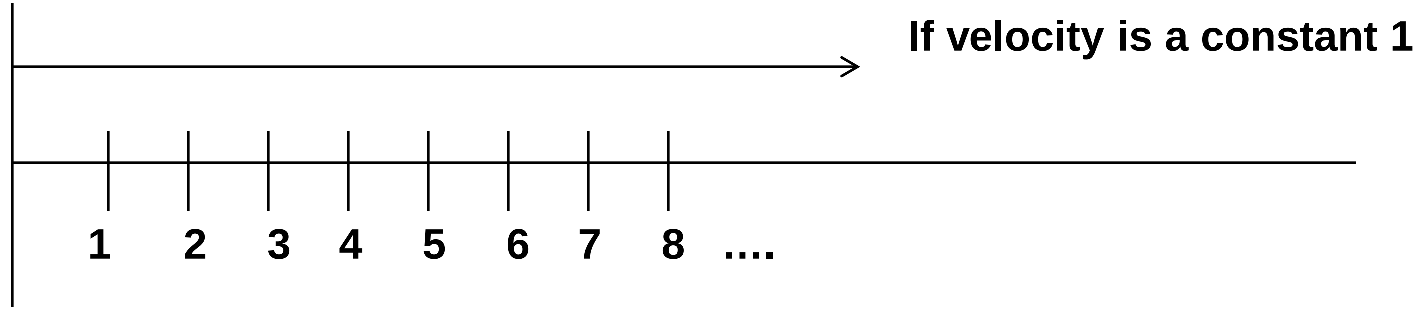
  has a lot of interesting uses.



If `F :: Behavior Float` (think function from time to Float) then `integral F z` is the area under the curve gotten by plotting `F` from `0` to `z`

Bouncing Ball revisited

- The bouncing ball has a constant velocity (either to the right, or to the left).
- Its position can be thought of as the integral of its velocity.



- At time t, the area under the curve is t, so the x position is t as well. If the ball had constant velocity 2, then the area under the curve is 2 * t, etc.

Bouncing Ball again

```
ex12 = wander x 0 yellow
       where xvel = 1 `stepAccum` (hit ->> negate)
             x = integral xvel
             left = x <=* -2.0 &&* xvel <*0
             right = x >=* 2.0  &&* xvel >*0
             hit = predicate (left ||* right)
```

Mouse Motion

- **The variable mm :: Event Vertex**
- **At every point in time it is an event that returns the mouse position.**

```
mouseDot =
   mm =>> \ (x,y) ->
            translate (constB x,constB y)
                                 dot

ex13 = paint red (dot `switch` mouseDot)
```

How does this work?

- Events are “real-time” actions that “happen” in the world. How do we mix Events and behaviors in some rational way.
- The Graphics Library supports a basic type that models these actions.

```
type Time = Float

data G.Event
  = Key       { char :: Char, isDown :: Bool }
  | Button    { pt :: Vertex, isLeft, isDown :: Bool }
  | MouseMove { pt :: Vertex }
  | Resize
  | Closed
deriving Show

type UserAction = G.Event
```

Type of Behavior

- In simple animations, a Behavior was a function from time. But if we mix in events, then it must be a function from time and a list of events.
- First try:

```
newtype Behavior1 a =
  Behavior1 ([(UserAction,Time)] -> Time -> a)
```

User Actions are time stamped. Thus the value of a behavior `(Behavior1 f)` at time `t` is, `f uas t`, where `uas` is the list of user actions.

Expensive because `f` has to “whittle” down `uas` at every sampling point (time `t`), to find the events it is interested in.

Solution

- **Sample at monotonically increasing times, and keep the events in time order.**

- **Analogy: suppose we have two lists xs and ys and we want to test for each element in ys whether it is a member of xs**

  - **inList :: [Int] -> Int -> Bool**
  - **result :: [Bool] -- Same length as ys**
  - **result1 :: map (inList xs) ys**

- **What’s the cost of this operation?**

- **This is analagous to sampling a behavior at many times.**

If xs and ys are ordered ...

```
result2 :: [Bool]
result2 = manyInList xs ys

manyInList :: [Int] -> [Int] -> [Bool]
manyInList [] _ = []
manyInList _ [] = []
manyInList (x:xs) (y:ys) =
      if y<x
         then manyInList xs (y:ys)
         else (y==x) : manyInList (x:xs) ys
```

Behavior: Second try

```
newtype Behavior2 a =
   Behavior2 ([(UserAction,Time)] ->
                   [Time] ->
                      [a])
```

- See how this has structure similar to the manyInList problem?

```
manyInList :: [Int] -> [Int] -> [Bool]
```

Refinements

```
newtype Behavior2 a =
   Behavior2 ([(UserAction,Time)] -> [Time] -> [a])

newtype Behavior3 a =
   Behavior3 ([UserAction] -> [Time] -> [a])

newtype Behavior4 a =
   Behavior4 ([Maybe UserAction] -> [Time] -> [a])
```

- **Final Solution**

```
newtype Behavior a
  = Behavior (([Maybe UserAction],[Time]) -> [a])
```

Events

```
newtype Event a =
   Event (([Maybe UserAction],[Time]) -> [Maybe a])
```

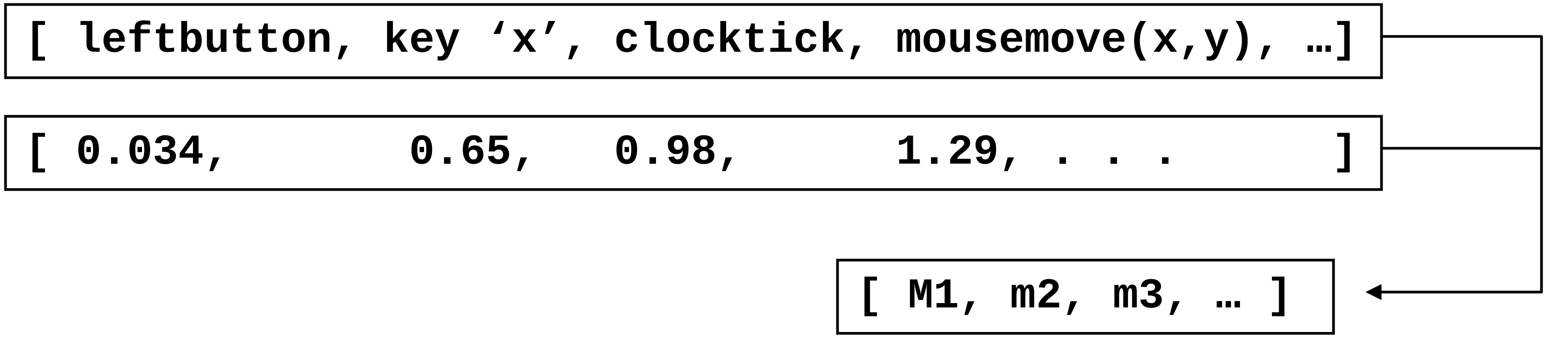
- **Note there is an isomorphism between the two types**

  `Event a` and `Behavior (Maybe a)`

- **We can think of an event, that at any particular time t, either occurs, or it doesn't.**
- **Exercise: Write the two functions that make up the isomorphism:**
  - **toEvent :: Event a -> Behavior (Maybe a)**
  - **toBeh :: Behavior(Maybe a) -> Event a**

Intuition

- **Intuitively it's useful to think of a `Behavior m` as transforming two streams, one of user actions, the other of the corresponding time (the two streams always proceed in lock-step) , into a stream of `m` things.**
- **User actions include things like**
  - **left and right button presses**
  - **key presses**
  - **mouse movement**
- **User Actions also include the "clock tick" that is used to time the animation.**



```
[ leftbutton, key 'x', clocktick, mousemove(x,y), …]

[ 0.034,       0.65,   0.98,      1.29, . . .      ]

[ M1, m2, m3, … ]
```

The Implementation

```
time :: Behavior Time
time = Behavior (\(_,ts) -> ts)
```

```
([ua1,ua2,ua3, …],[t1,t2,t3, …]) --->
                [t1, t2, t3, …]
```

```
constB :: a -> Behavior a
constB x = Behavior (\_ -> repeat x)
```

```
([ua1,ua2,ua3, …],[t1,t2,t3, …]) --->
                [x, x, x, …]
```

Simple Behaviors

```
red, blue :: Behavior Color
red    = constB Red
blue   = constB Blue

lift0 :: a -> Behavior a
lift0 = constB
```

Notation

- We often have two versions of a function:

```
xxx :: Behavior a -> (a -> b) -> T b

xxx_ :: Behavior a ->  b -> T b
```

- And two versions of some operators:

```
(=>>) :: Event a -> (a->b) -> Event b

(->>) :: Event a -> b -> Event b
```

Lifting ordinary functions

```
($*) :: Behavior (a->b) -> Behavior a -> Behavior b
Behavior ff $* Behavior fb
  = Behavior (\uts -> zipWith ($) (ff uts) (fb uts)
        where f $ x = f x
```

```
([t1,t2,t3, …],[f1,f2,f3, …]) --->
([t1,t2,t3, …],[x1,x2,x3, …]) --->
([t1,t2,t3, …],[f1 x1, f2 x2, f3 x3, …]
```

```
lift1 :: (a -> b) -> (Behavior a -> Behavior b)
lift1 f b1  = lift0 f $* b1

lift2 :: (a -> b -> c) ->
            (Behavior a -> Behavior b -> Behavior c)
lift2 f b1 b2  = lift1 f b1 $* b2
```

Button Presses

```
data G.Event
  = Key       { char :: Char, isDown :: Bool }
  | Button    { pt :: Vertex, isLeft, isDown :: Bool }
  | MouseMove { pt :: Vertex }
```

```
lbp :: Event ()
lbp = Event (\(uas,_) -> map getlbp uas)
   where getlbp (Just (Button _ True True)) = Just ()
         getlbp _                           = Nothing
```

```
([Noting, Just (Button …), Nothing, Just(Button …), …],
 [t1,t2,t3, …]) --->
            [Nothing, Just(), Nothing, Just(), …]
```

```
Color0 = red `switch` (lbp --> blue)
```

Key Strokes

```
key :: Event Char
key = Event (\(uas,_) -> map getkey uas)
      where getkey (Just (Key ch True)) = Just ch
            getkey _                    = Nothing
```

```
([leftbut, key ‘z’ True, clock-tick, key ‘a’ True …],
 [t1,      t2,           t3,         t4,          …])

          --->

           [Nothing, Just ‘z’, Nothing, Just ‘a’, …]
```

Mouse Movement

```
mm :: Event Vertex
mm = Event (\(uas,_) -> map getmm uas)
     where getmm (Just (MouseMove pt))
                          = Just (gPtToPt pt)
           getmm _        = Nothing
```

```
([Noting, Just (MouseMove …), Nothing, Just(MouseMove …), …],
 [t1,t2,t3, …]) --->
           [Nothing, Just(x1,y1), Nothing, Just(x2,y2), …]
```

```
mouse :: (Behavior Float, Behavior Float)
mouse = (fstB m, sndB m)
          where m = (0,0) `step` mm
```

```
( (uas,ts) --> [x1,x2, …],
  (uas,ts) --> [y1, y2, …] )
```

Behavior and Event Combinators

```
switch :: Behavior a -> Event (Behavior a) -> Behavior a
Behavior fb `switch` Event fe =
  memoB
    (Behavior
       (\uts@(us,ts) -> loop us ts (fe uts) (fb uts)))
 where loop (_:us) (_:ts) ~(e:es) (b:bs) =
      b : case e of
            Nothing -> loop us ts es bs
            Just (Behavior fb')
               -> loop us ts es (fb' (us,ts))
```

```
([Noting,Just (Beh [x,y,...] …),Nothing,Just(Beh [m,n,…])…],
 [t1,t2,t3, …]) --->
            [fb1, fb2, x, y, m, n …]
```

Event Transformer (map?)

```
(=>>) :: Event a -> (a->b) -> Event b

Event fe =>> f = Event (\uts -> map aux (fe uts))
  where aux (Just a) = Just (f a)
        aux Nothing  = Nothing

(->>) :: Event a -> b -> Event b

e ->> v = e =>> \_ -> v
```

```
([Noting, Just (Ev x), Nothing, Just(Ev y), …] --> f -->
            [Nothing, Just(f x), Nothing, Just(f y), …]
```

withElem

```
withElem  :: Event a -> [b] -> Event (a,b)
```

Infinite list

```
withElem (Event fe) bs
 = Event (\uts -> loop (fe uts) bs)
      where loop (Just a  : evs) (b:bs)
                = Just (a,b) : loop evs bs
            loop (Nothing : evs)    bs
                = Nothing    : loop evs bs

withElem_ :: Event a -> [b] -> Event b
withElem_ e bs = e `withElem` bs =>> snd
```

```
([Noting, Just x, Nothing, Just y, …]) ---> [b0,b1,b2,b3, …] ->
            [Nothing, Just(x,b0), Nothing, Just(y,b1), …]
```

Either one event or another

```
(.|.) :: Event a -> Event a -> Event a

Event fe1 .|. Event fe2
  = Event (\uts -> zipWith aux (fe1 uts) (fe2 uts))
      where aux Nothing  Nothing  = Nothing
            aux (Just x) _        = Just x
            aux _        (Just x) = Just x
```

```
([Noting, Just x, Nothing, Just y, …]) --->
 [Nothing, Just a, Just b, Nothing, …] --->
           [Nothing, Just x, Just b, Just y, …]
```

Snapshot

```
snapshot :: Event a -> Behavior b -> Event (a,b)

Event fe `snapshot` Behavior fb
  = Event (\uts -> zipWith aux (fe uts) (fb uts))
      where aux (Just x) y = Just (x,y)
            aux Nothing  _ = Nothing

snapshot_ :: Event a -> Behavior b -> Event b
snapshot_ e b = e `snapshot` b =>> snd
```

```
[Nothing, Just x, Nothing, Just y, …] --->
[b1,     b2,     b3,      b4,     …] --->
        [Nothing, Just(x,b2), Nothing, Just(y,b4), …]
```

step and stepAccum

```
step :: a -> Event a -> Behavior a
a `step` e = constB a `switch` e =>> constB
```

```
X1 -> [Nothing, Just x2, Nothing, Just x3, …] --->
      [x1,     x1,      x2,      x2,     x3, ...]
```

```
stepAccum :: a -> Event (a->a) -> Behavior a
a `stepAccum` e = b
   where b = a `step`
                 (e `snapshot` b =>> uncurry ($))
```

```
X1 -> [Noting, Just f, Nothing, Just g, …] --->
      [x1,     x1,     f x1,    (f x1), g(f x1), ...]
```

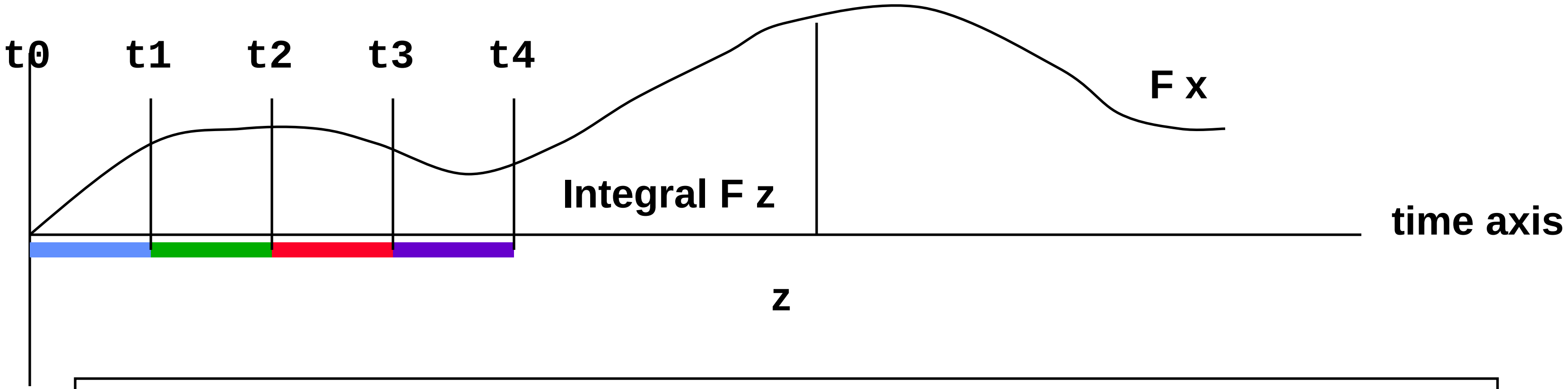
predicate

```
predicate :: Behavior Bool -> Event ()

predicate (Behavior fb)
  = Event (\uts -> map aux (fb uts))
    where aux True  = Just ()
          aux False = Nothing
```

```
[True,   True,   False,   True,   False, …] --->
[Just(), Just(), Nothing, Just(), Nothing, ...]
```

integral

```
integral :: Behavior Float -> Behavior Float
integral (Behavior fb)
  = Behavior (\uts@(us,t:ts) ->
                0 : loop t 0 ts (fb uts))
      where loop t0 acc (t1:ts) (a:as)
                  = let acc' = acc + (t1-t0)*a
                     in acc' : loop t1 acc' ts as
```



([ua0,ua1,ua2,ua3, …],[t0,t1,t2,t3, …]) --->

[0, Area t0-t1, Area t0-t2, Area t0-t3, …]

Putting it all together

```
reactimate :: String -> Behavior Graphic -> IO ()
reactimate title franProg
  = runGraphics $
    do w <- openWindowEx title (Just (0,0)) (Just (xWin,yWin))
              drawBufferedGraphic
       (us,ts,addEvents) <- windowUser w
       addEvents
       let drawPic (Just g) =
             do setGraphic w g
                quit <- addEvents
                if quit
                  then return True
                  else return False
           drawPic Nothing  = return False
       let Event fe = sample `snapshot_` franProg
       run drawPic (fe (us,ts))
       closeWindow w
```

```
where
    run f (x:xs) = do
      quit <- f x
      if quit
        then return ()
        else run f xs
    run f [] = return ()
```

The Channel Abstraction

```
(us,ts,addEvents) <- windowUser w
```

- `us`, and `ts` are infinite streams made with channels.
- A Channel is a special kind of abstraction, in the multiprocessing paradigm.
- If you “pull” on the tail of a channel, and it is null, then you “wait” until something becomes available.
- `addEvents :: IO ()` is a action which adds the latest user actions, thus extending the streams `us` and `ts`

Making a Stream from a Channel

```
makeStream :: IO ([a], a -> IO ())
makeStream = do
  ch <- newChan
  contents <- getChanContents ch
  return (contents, writeChan ch)
```

A Reactive window

```
windowUser :: Window -> IO ([Maybe UserAction], [Time], IO Bool)
windowUser w
  = do (evs, addEv) <- makeStream
       t0 <- timeGetTime
       let addEvents =
             let loop rt = do
                   mev <- maybeGetWindowEvent w
                   case mev of
                     Nothing -> return False
                     Just e  -> case e of
                        Key ' ' True -> return True
                        Closed -> return True
                        _ -> addEv (rt, Just e) >> loop rt
             in do t <- timeGetTime
                   let rt = w32ToTime (t-t0)
                   quit <- loop rt
                   addEv (rt, Nothing)
                   return quit
       return (map snd evs, map fst evs, addEvents)
```

The “Paddle Ball” Game

```
paddleball vel = walls `over` paddle `over` ball vel

walls = let upper = paint blue
                        (translate ( 0,1.7) (rec 4.4 0.05))
            left  = paint blue
                        (translate (-2.2,0) (rec 0.05 3.4))
            right = paint blue
                        (translate ( 2.2,0) (rec 0.05 3.4))
        in upper `over` left `over` right

paddle = paint red
        (translate (fst mouse, -1.7) (rec 0.5 0.05))

x `between` (a,b) = x >* a &&* x <* b
```

The “reactive” ball

```
pball vel =
 let xvel    = vel `stepAccum` xbounce ->> negate
     xpos    = integral xvel
     xbounce = when (xpos >*  2 ||* xpos <* -2)
     yvel    = vel `stepAccum` ybounce ->> negate
     ypos    = integral yvel
     ybounce = when (ypos >* 1.5
               ||* ypos      `between` (-2.0,-1.5) &&*
                   fst mouse `between`
                          (xpos-0.25,xpos+0.25))
➢ in paint yellow (translate (xpos, ypos) (ell 0.2 0.2))

main = test (paddleball 1)
```