Elements of Programming Languages
Assignment 3: Functional Reactive Animation
Version 1.1 (October 31)
Due: November 23, 12pm

October 31, 2022

1 Introduction

Functional Reactive Programming [1] (or FRP) is an approach to programming reactive systems, such as animations, user interfaces, and music. A number of *domain-specific languages* have been proposed based on FRP, with one of the earliest being Fran, a language for Functional Reactive Animation, in Haskell.

A domain-specific language (DSL) is a programming language which is specialised to some problem domain. DSLs are ubiquitous: perhaps the best-known example is SQL for constructing database queries, but the functional approach to implementing DSLs has seen much use in industry, in particular for financial modelling[2]. Ideas from FRP have influenced a number of domain-specific languages, suck as Scala's Akka framework, as well as some general-purpose ones, for example the Flapjax and Elm web programming languages (although Elm's original FRP-based approach has been superseded.)

In this assignment, we will investigate two ways of implementing a domain specific language for animations based on FRP. The first involves implementing an *embedded* domain specific language (or EDSL), where a DSL is embedded within some host language, and can therefore make use of the syntax, semantics, flow control, and binding structures of the host language.

The second involves implementing a *standalone* DSL, $\lambda_{\mathsf{RABBIT}}$. Implementing a standalone DSL means that the language constructs must be implemented from scratch, but the language designer has more control over features such as the syntax, evaluation strategy, and type system.

This assignment is due November 23, at 12pm.

Please read over this handout carefully and look over the code before beginning work, as some of your questions may be answered later. Please let us know if there are any apparent errors or bugs. We will try to update this handout to fix any major problems and such updates will be announced to the course mailing list. The handout is versioned and the most recent version should always be available from the course web page.

2 Constructs in FRP

The idea behind FRP is to specify dynamic phenomena *declaratively* (by describing what should happen) rather than by defining and maintaining the current state and repeatedly updating it *imperatively* as one would naturally do in a non-functional language.

[1] https://en.wikipedia.org/wiki/Functional_reactive_programming

[2] see http://www.timphilipwilliams.com/slides/HaskellAtBarclays.pdf and http://www.dmst.aueb.gr/dds/pubs/jrnl/2008-JFP-ExoticTrades/html/FSNB08.pdf

```
1 moveXY(time,time,read("turtle"))
```

Figure 1: Moving the turtle diagonally in λ_{RABBIT}

```
1 moveXY(pure(\ x : Int -> x * 20) <*> time,pure(100),read("turtle"))
```

Figure 2: Moving the turtle faster, horizontally

Figure 1 shows a simple λ_{RABBIT} program that moves an image (in this case a turtle) from the centre of the canvas towards the upper right corner. In FRP, *signals* are used to represent time-varying data. For example, the `time` signal always provides the current time. Its type is `signal[int]` where `int` is the type of our abstraction of environment times. For our purposes times will just be integers representing number of steps since the start of the animation. The `read(img)` signal takes a file containing an image filename and loads it, producing a signal of type `signal[frame]` that draws the image at the origin, where `frame` is the type of 2-dimensional images. Finally, the `moveXY` operation takes three signals: two `signal[int]` signals giving the x and y coordinates, and a third signal `signal[frame]` that provides an animation. The result of `moveXY` is a signal `signal[frame]` in which each frame is shifted by the current x and y values of the first two arguments. Putting these constructs together, the above example loads the turtle image, and renders it at position (0,0), then (1,1), and so on, moving it along the diagonal.

The constructs mentioned so far are somewhat inflexible: for example `time` is the only way to produce a signal that can be used as the first or second argument to `moveXY`. We do not even have a way to create a constant signal whose value is always some integer, say 100. This would make it possible to move the turtle horizontally only, using the time for the x-axis and a constant for the y-axis. The `pure(e)` signal is a signal whose value is the constant value of e. This allows us to take any ordinary value and turn it into a signal (this includes function values). Moreover, we don't currently have any general way to transform the output of a signal; although the `moveXY` signal provides a special case of such a transformation. In general, we might like to be able to apply an ordinary function $f : a \to b$ to a signal `signal`$[a]$ to get a signal `signal`$[b]$. Slightly more generally we could even allow for applying a *function signal* f : `signal`$[a \to b]$ to an argument signal `signal`$[a]$ to get a signal `signal`$[b]$. (This is more general because we can always treat an ordinary function as a function signal using `pure`). Our next construct, called *lifting*, permits this. Specifically if f : `signal`$[a \to b]$ and x : `signal`$[a]$ then f `<*>` x : `signal`$[b]$. The result of lifting is the signal that provides current value of f applied to the current value of x.

Using these two new constructs, Figure 2 shows how to make the turtle move horizontally, and much faster: in the x dimension the turtle moves 20 steps in each frame, accomplished by lifting the fucntion $\lambda x.x * 20$ applied to time, while the y component is a constant.

In the above example we might notice that programming with signals becomes a little awkward because pure values need to be converted to signals using `pure`, and to adjust the behaviour of an existing signal we need to define the adjustment as a pure function and apply it using `<*>`. We'd like to be able to write an adjustment more directly for example as time $*$ 20. This however does not typecheck since time has type `signal[int]` while $*$ expects two arguments of type `int`. There are a number of ways of getting around this, such as introducing variants of the primitive operations that work on signals, using subtyping (considering *int* to be a subtype of *signal*[*int*]), or using overloading or implicit conversions (for example if we attempt to multiply an integer signal and an integer, convert the latter to a signal first). We will take an alternative, slightly more verbose but also more predictable approach, by allowing *signal blocks*.

Figure 3 illustrates a simple use of a signal block (a more complete description is in Section 6).

```
1 signal { moveXY(time * 20, 100, read("turtle")) }
```

Figure 3: Signal blocks

```
1  signal { moveXY(time * 50 - 500, 200, read("turtle")) <+>
2           moveXY(when(time < 5, time * 100 - 500, 0), -200, read("rabbit")) }
```

Figure 4: Race between turtle and hare

Within such a block, numbers and operators are implicitly treated as constant signals, while uses of signal operations such as `moveXY` or `time` are treated specially. In this assignment you are asked to implement a translation from the above convenient notation to the more explicit form shown in Figure 2.

We haven't used all of the available operations in the above examples, so for easy reference we summarize them below.

- time: the current time
- pure e: convert a plain value into a constant signal
- e_1 <*> e_2: apply a signal function to a signal providing argument values
- moveXY(e_1, e_2, e_3): apply a transformation using signals e_1 and e_2 for the x and y values to the animation e_3 (these are deltas to the current origin of the animation).
- scale(f, e): scale the animation e by the scale factor given by signal f
- rotate(r, e): rotate the animation e by the radius given by signal r
- blank: an empty animation
- e_1 <+> e_2: superimpose two animations e_1 and e_2.
- when(e_1, e_2, e_3): using Boolean signal e_1, choose between the value of e_2 or e_3

The additional operations *over* and *when* are useful for combining animations by superposition or using a conditional (Boolean) signal to choose between them, respectively. The *blank* signal is useful as a neutral value for *over* or *when*. Figure 4 illustrates their use to animate a race between the turtle and the rabbit: the rabbit is faster but stops at some point and the turtle catches up. The mysterious number `-500` in the code is due to our coordinates system which will be explained in detail in Section 3.2.

Of course, these small examples are rather uninteresting! Surprisingly, given variables, recursion, list, and higher-order functions, we can quickly make some more interesting animations. The program `overAll.rab` later in this document shows a simple example, where we build a list of pairs of images and speeds, and apply a general function to "race" several animations, using the `over` operation to superimpose animations. You do not need to understand the code in depth—we will introduce the language constructs in more detail in Section 6—but it is useful to see what can be done! We include several in the `examples/` directory, and encourage you to explore some λ_{RABBIT} programs of your own.

3 Included Code: a Simple Graphics and FRP Library

To help you get started, we provide `AnimatedGif.scala`, a simple graphics library to wrap around sophisticated image-processing library called `scrimage` and Java's `Graphics2D` library, and `FRP.scala`, a simple FRP library based on signals. You should not need to use any Java/Scala graphics or FRP libraries directly. We have already imported the needed library interfaces in the exercise files. We provide more detailed description of these two libraries in Section 3.1 and Section 3.2.

3.1 Functional Reactive Programming Primitives

We provide `FRP.scala`, a simple library for Functional Reactive Programming based on signals. You do not need to understand the implementation details of this library as it involves some advanced Scala features. [3] The interfaces that might be used are as follows:

[3] If you are interested, you can have a look at this series of blogs.

- `Signal[T]` is the type of a time-varying value of type `T`.
- `type Time = Int` is the type of environment time. In our simplified case, time are modelled as integers.
- `envtime: Var[Time]` is the signal that produces environment time.
- `pure[A](a: A): Signal[A]` lifts a value of type `A` to a constant signal, i.e. a signal that always produces the same value at any time.
- `app[A, B](f: Signal[A => B])(t: Signal[A]): Signal[B]` applies a function `f` wrapped in a signal to a signal parameter, and then returns another signal. Given that signals are time-varying values in our simplified situation, intuitively, $app(f, a)$ means that given time t, we get the function value f_t and parameter value a_t at time t, and apply f_t to a_t to get the result at time t. Thus, the result is also a time-varying value.
- `lift1, lift2, lift3` are three helper functions wrapping `pure` and `app`.

The `pure` and `app` form the so-called applicative interfaces. We could write function applications with arbitrary number of arguments using them. For example, say we have a function `f:A=>B=>C=>D` and three arguments `a:Signal[A], b:Signal[B], c:Signal[C]`, we could get a signal value of type `Signal[D]` by writing `app(app(app(pure(f), a), b), c)`.

We have already provided three functions `lift1, lift2, lift3` for you to apply a function to multiple signals. Note that the function to be applied is the last parameter of these lifting functions. It is used to deal with a type inference problem of Scala that might appear when implementing the `when` operation of DSL. You can try `lift3` if your code written directly using the applicative inferface does not pass the type checking.

3.2 Creating Animated GIFs

We provide `AnimatedGif.scala`, which is a simple graphics library to wrap around sophisticated image-processing library called `scrimage`. It uses `GraphicsCanvas.scala`, another simple graphics library wrapping around Java's `Graphics2D` library, to draw pictures.

```
1 case class Picture( var name: String, var x: Integer = 0, var y: Integer = 0
2                   , var scaleFactor: Double = 1, var angle: Double = 0)
```

The co-ordinate system used by `AnimatedGif.scala` is the standard Cartesian co-ordinate system in a plane. Co-ordinates are integer values, where each integer refers to a pixel. The canvas is 1000×1000 by default and the centre of the canvas is $(0, 0)$. Figure 5 shows example co-ordinates for the default canvas.

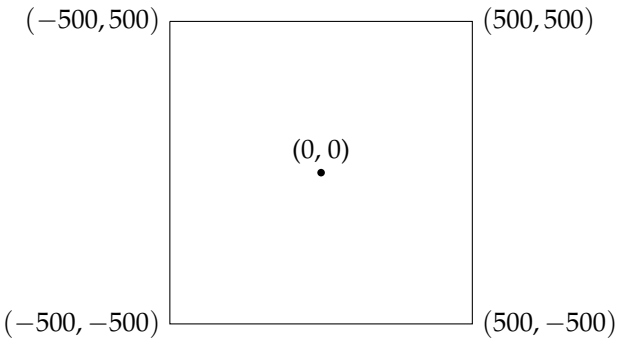

Figure 5: Co-ordinate System

We assume that the angles are measured clockwise relative to the positive y-axis, so that angle 0 corresponds to no rotation. (This choice is somewhat arbitrary.) Figure 6 shows how angles are measured. You may want to refresh your memory of basic trigonometry, for example to calculate x_1 and y_1 below from θ and the length d of the line from $(0, 0)$ to (x_1, y_1), we can obtain x_1 as

$d \times sin(\theta)$ and y_1 as $d \times \cos(\theta)$. Scala's *sin* and *cos* functions are imported from Java's math library in the files where you will need them.

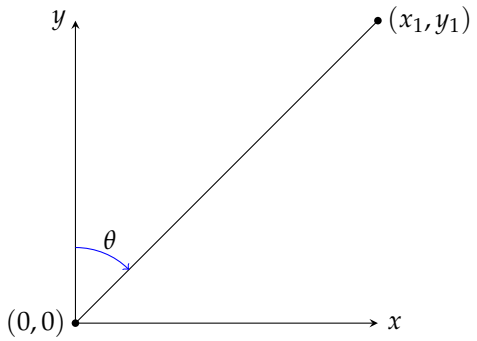

Figure 6: Angles.

Pictures are defined as a case class with file name, x-axis co-ordinate, y-axis co-ordinate, scale factor, and rotation angle (in **radians**). The default parameters put the picture in the centre of the canvas with no scale and rotation.

Frames are defined as lists of pictures, and animations are time-varying frames, i.e. `Signal[Frame]`. We do not define a type alias for animations as it is rarely used. The order of pictures in the frame determines the order of drawing, i.e. if picture `p1` appears later than `p2` in the frame, then `p2` should appear on top of `p1` if there is any overlap.

```
1 type Frame = List[Picture]
```

There are also some global animation configurations. You do not need to change them.

- `val width = 1000` is the canvas width
- `val height = 1000` is the canvas height
- `val unit = 0.1` means that 1 unit of time is equal to 0.1 seconds in real world.

The only function that you may need to use is `generateGif(frames, timeEnd, filename)` which generates an animated GIF with `timeEnd` frames from an animation `frames` and saves it at a path given by `filename`.

```
1 def generateGif(frames: Signal[Frame], timeEnd: Time, filename: String): Unit
```

4 Getting started

`Assn3.zip` contains a number of starting files; use the `unzip` command to extract them. We provide the following Scala files.

- `FRP.scala`, giving an implementation of the FRP primitives. You should not change this file.
- `AnimatedGif.scala`, providing techniques for createing animated GIFs. You should not change this file.
- `RabbitEDSL.scala`, containing the supporting code and template for the Rabbit EDSL which you are to implement in Exercise 1.
- `RabbitSyntax.scala`, containing the class definitions for the abstract syntax of Rabbit. You should not change this file.
- `RabbitParser.scala`, containing code for a simple parser for Rabbit programs. You should not change this file.
- `RabbitStandalone.scala`, containing the supporting code and template for the standalone implementation of $\lambda_{\mathsf{RABBIT}}$ which you are to implement in Exercises 2–7.

We include several example programs in a directory `examples`. Some are defined using the EDSL approach in `RabbitEDSL.scala`, and `run_edsl.sh` will run these examples and generate some sample GIF outputs. Others are provided as files for use with the standalone DSL, and can be run using `run.sh` or `test.sh`.

For convenience we will use SBT (the Scala Build Tool) whcih is available on DICE machines at version 1.4.9. I you are running Scala on a non-DICE machine where you can install SBT yourself, this is encouraged; other more recent versions should also work. Among other things, SBT deals with the logistics of downloading and linking together libraries that ou code depends on so you don't have to.

The most useful ways to call SBT from the command line are:

```
sbt compile    -- compiles everything
sbt package    -- ... and builds jar file
sbt run        -- ... and runs main class
sby "runMain Class arg1 ... argn"
  -- ... or runs a specified main class with specified arguments
```

If there are several classes providing a `main` method entry point, the `sbt run` command will ask interactively which one to run. To avoid this interactive choice, the `runMain` command can be used, which takes a class name and any command line arguments (the quotation marks are important).

SBT should always be invoked from the root directory of the assignment code, e.g. `Assn3`, where the `build.sbt` file is. (You should not need to change this file.) **Be aware that the first time you invoke SBT may take several minutes, becasue it initializes and downloads many libraries.**

SBT also has an unforutnate interaction with networked file systems such as AFS which is used on DICE machines. As a result it asks whether to "create a new server" whenever you run it on DICE. The compile script below avoids this behavior.

We also provide several scripts (which should work on a DICE, Linux or MacOS system):

- `compile.sh` compiles the code and pakcages it as a jar file (in a place where the remaining scripts will assume they can find it).
- `run_edsl.sh` runs the EDSL code using your implementation
- `sample_edsl.sh` runs the EDSL code using the sample solution
- `run.sh <file> <gif>` runs your standalone interpreter on a Rabbit file, using your EDSL implementation, and saving the resulting animation if any in the provided GIF filename
- `run_sample.sh <file> <gif>` works like `run.sh` but runs your standalone interpreter on a Rabbit file, using the sample solution's EDSL implementation. (This script should be helpful for working on exercises 2–7 before you have a working implementation of exercise 1.)
- `sample.sh <file> <gif>` works like `run.sh` but uses the sample solution both to run the interpreter and to run the eresulting EDSL code.
- `test.sh <file>` runs the standalone interpreter on a Rabbit file and prints the end result value. (This script should be helpful for testing non-animation-related features of Rabbit.)

In addition a JAR file `Assn3Solution.jar` containing a working implementation is provided. The scripts `sample.sh`, `sample_edsl.sh` and `run_sample.sh` use this.

If you are using a non-DICE system, such as Windows, these scripts may not work, but you can look at the contents to work out what you need to do instead.

4.1 Objectives

The rest of this handout defines exercises for you to complete, building on the partial implementation in the provided files. You may add your own function definitions or other code, but please use the existing definitions/types for the functions we ask you to write in the exercises, to simplify automated testing we may do. You should not need to change any existing code other than filling in definitions of functions as stated in the exercises below.

Your solutions may make use of Scala library operations, such as the list and list map operations that have been covered in previous assignments.

This assignment relies on material covered up to Lecture 14 (November 10). The two parts of this assignment are independent and can be attempted in either order.

This assignment is graded on a scale of 100 points, and amounts to 25% of your final grade for this course. Your submissions will be marked and returned with feedback within 2 weeks if they are received by the due date.

Unlike the other two assignments, which were for feedback only, you must work on this assignment individually and not with others, in accordance with University policy on academic conduct. Please see the course web page for more information on this policy.

Submission instructions You should submit a single ZIP file, called `Assn3.zip`, with the missing code in the two main Scala files filled in as specified in the exercises in the rest of this handout. Any files in the submission other than `RabbitEDSL.scala` and `RabbitStandalone.scala` will be ignored so please be careful not to make changes to any of the other files in your solution. Please submit the ZIP file through Learn following the instructions on the Learn page for this assignment. The submission deadline is 12pm on November 23.

5 Part 1: An Embedded Domain-Specific Language

In this section, you will implement Functional Reactive Animation as an embedded domain-specific language, using Scala as the host language. Embedded domain-specific languages allow the syntax and semantics to be "borrowed" from the host language, meaning that they do not need to be implemented separately.

The EDSL is contained within `RabbitEDSL.scala`.

5.1 Compiling and Running the EDSL

Compiling the DSL The `./compile.sh` command will compile the DSL, as well as `AnimatedGif`, `FRP` and `GraphicsCanvas` if required.

Running the DSL The `RabbitEDSL.scala` file contains a definition `val toRun = List(...)`, containing a list of (`RabbitAnimation`, `Duration`, `Filename`) tuples. To run the file, run `./run_edsl.sh`, or alternatively:

```
sbt "runMain Assignment3.RabbitEDSL.Assignment3Embedded"
```

This will execute all entries in the `toRun` list and save them to their corresponding filenames.

The `RabbitEDSL` file includes an object `Testing` that "implements" `RabbitDSL` by printing a message to the terminal whenever one of its methods is called. This may be helpful for testing your solution to Exercise 1 below. Initially, the code contains these lines:

```
import Testing._
// import RabbitDSLImpl._
```

You should comment out the first line and uncomment the second when you are ready to test your solution to Exercise 2.

5.2 EDSL Definition

The interface that you will implement is as follows:

```
1  trait RabbitDSL {
2    type RabbitAnimation[T]
3    def time: RabbitAnimation[Time]
4    def read(name: String): RabbitAnimation[Frame]
5    def blank: RabbitAnimation[Frame]
```

```
6     def pure[A](t: A): RabbitAnimation[A]
7     def app[A, B](f: RabbitAnimation[A => B], t: RabbitAnimation[A]) : RabbitAnimation[B]
8     def when[A]( b: RabbitAnimation[Boolean], t1: RabbitAnimation[A]
9                , t2: RabbitAnimation[A]): RabbitAnimation[A]
10    def moveXY( dx: RabbitAnimation[Int], dy: RabbitAnimation[Int]
11              , a: RabbitAnimation[Frame]): RabbitAnimation[Frame]
12    def scale(factor: RabbitAnimation[Double], a: RabbitAnimation[Frame])
13             : RabbitAnimation[Frame]
14    def rotate(angle: RabbitAnimation[Double], a: RabbitAnimation[Frame])
15              : RabbitAnimation[Frame]
16    def over(x: RabbitAnimation[Frame], y: RabbitAnimation[Frame]) : RabbitAnimation[Frame]
17    def runRabbitAnimation[T](term: RabbitAnimation[T]): Signal[T]
18    def saveToFile( anim: RabbitAnimation[Frame], timeEnd: Time , filename: String) : Unit
```

Here, `RabbitAnimation[T]` is an abstract type which you will have to define in your implementation. Note that it is parameterized by the type parameter `T` which means that the type of the result value of the DSL program is `Signal[T]` as shown by `runRabbitAnimation`. The constructs are as follows:

- `time` the signal of environment time
- `read(name)` an operation which creates a constant animation from a picture that has name `name`
- `blank`: a blank animation with no pictures
- `pure(t)`: an operation which lifts the value `t` to a constant signal
- `app(f, t)`: an operation which applies a signal function `f` to a signal `t`. We also provide an infix operator `f <*> t` for `app(f, t)`, which you do not need to understand and should not change.
- `when(b, t1, t2)`: an operation which returns signal `t1` when the value of signal `b` is true, o therwise returns signal `t2`
- `moveXY(x, y, a)`: an operation which moves the animation `a` according to the changes in coordinates given by signal `x` and signal `y`. It moves every picture in the animation. For example, if at time t the i-th picture of the animation `a` is at (x_t^i, y_t^i), and the values of signals `x` and `y` are x_t and y_t, then after moving this picture is at $(x_t^i + x_t, y_t^i + y_t)$.
- `scale(factor, a)`: an operation which scales the animation `a` according to the scale factor `factor`. It scales every picture in the animation. It scales not only the size of pictures, but also their co-ordinates. For example, if at time t the i-th picture of the animation `a` is at (x_t^i, y_t^i) and has scale factor f_t^i, and the value of the signal `factor` is f_t, then after scaling this picture is at $(x_t^i \times f_t, y_t^i \times f_t)$ and has scale factor $f_t^i \times f_t$.
- `rotate(angle, a)`: an operation which rotates the animation `a` according to the radius `angle`. It rotates every picture in the animation. It rotates not only the pictures themselves, but also their co-ordinates. For example, if at time t the i-th picture of the animation `a` is at (x_t^i, y_t^i) and has angle r_t^i, and the value of the signal `angle` is r_t, then after scaling this picture is at $(x_t^i \times \cos(r_t) + y_t^i \times \sin(r_t), -x_t^i \times \sin(r_t) + y_t^i \times \cos(r_t))$ and has angle $r_t^i + r_t$.
- `over(a1, a2)` an operation which combines two animations. When overlapping, the pictures in the first animation `a1` should be on top of the second animation `a2`. We also provide code for an infix operator `a1 <+> a2` for `over(a1, a2)`, which you do not need to understand and should not change.
- `runRabbitAnimation(term)` runs the DSL program `term` and return the result.
- `saveToFile(anim, timeEnd, filename)` takes an animation `anim`, generates an animated GIF of duration `timeEnd` from it, and saves the resulting GIF to the location given by `filename`. We have provided the implementation of this operation.

For example, using the DSL, we can write a program to make an animation of a race between a turtle and a rabbit:

```
moveXY(pure({x:Int => x*50-500}) <*> time, pure(200), read("turtle")) <+>
moveXY(when( pure({x:Int => x < 5}) <*> time
           , pure({x:Int => x*100-500}) <*> time
           , pure(0)) , pure(-200), read("rabbit"))
```

You are to define an implementation of the `RabbitDSL` trait. The rest of this section gives additional suggestions regarding how to proceed.

Exercise 1. *Define an object* `RabbitDSLImpl` *extending* `RabbitDSL` *that provides definitions for all of the components.*

[40 marks]

5.3 EDSL Implementation Strategies

You are free to implement the interface as you see fit. Key to your implementation is which type you will use as a concrete instantiation of the `RabbitAnimation` type, which will generally follow one of two patterns known as either a *deep* or a *shallow* embedding.

Deep Embeddings A *deep embedding* encodes each language construct as a node in an abstract syntax tree, and "executes" the DSL by acting as an interpreter for each DSL construct. An example structure is:

```
object DeepRabbitDSL extends RabbitDSL {
  type RabbitAnimation[T] = RabbitTerm[T]

  // AST Definition
  abstract class RabbitTerm[T]
  case object RabbitTime extends RabbitTerm[Time]
  case class RabbitOver(x: RabbitTerm[Animation], y: RabbitTerm[Animation])
    extends RabbitTerm[Animation]
  ...

  // Language constructs create AST nodes
  def time = RabbitTime
  def over(x: RabbitAnimation[Animation], y: RabbitAnimation[Animation]) = RabbitOver(x, y)
  ...

  // runRabbitAnimation interprets the AST
  def runRabbitAnimation[T](term: RabbitAnimation[T]): Signal[T] = term match {
    case RabbitTime => envtime
    case RabbitOver(x, y) => {
      val sx = runRabbitAnimation(x)
      val sy = runRabbitAnimation(y)
      lift2(sx)(sy)({(x:Frame) => (y:Frame) => y ++ x})
    }
    ...
  }
}
```

Note that we also give a type parameter `T` to the case class `RabbitTerm[T]`. It represents the type of the result value after interpretation as shown by the type of `runRabbitAnimation`.

Shallow Embeddings A *shallow embedding* implements each construct as a function or value. For `RabbitDSL`, since it has no context or state, we can directly implement constructs as values of type `Signal[T]`. For example, since `time: RabbitAnimation[Time]`, and the type `Time` is defined as `Int`, we need to provide an appropriate integer signal, e.g. `envtime`. Thus, the `runRabbitAnimation`

function has a very simple implementation, we simply return the input as it is already of type `Signal[T]`. However, this means that the work of building an appropriate value of type `Signal[T]` needs to be done in each of the operations.

```
object ShallowRabbitDSL extends RabbitDSL {
  type RabbitAnimation[T] = Signal[T]

  def time = envtime
  def over(x: RabbitAnimation[Frame], y: RabbitAnimation[Frame]) = {
    lift2(x)(y)({(x:Frame) => (y:Frame) => y ++ x})
  }
  ...

  def runRabbitAnimation[T](term: RabbitAnimation[T]): Signal[T] = term
}
```

6 Part 2: A Standalone DSL

In this part of the assignment you will implement a *standalone* DSL variant of Rabbit called λ_{RABBIT}. "Standalone" means that we are not re-using Scala as a host language, so we need to parse in λ_{RABBIT} programs, typecheck them, implement substitution, expand syntactic sugar, and finally evaluate the programs. Luckily, we have done some of this (e.g. parsing) for you, and the concrete and abstract syntax of the Rabbit standalone DSL is based on the Giraffe language from Assignment 2, so you should already be familiar with it. (This also means that we will assume familiarity with the contents of Assignment 2, and not explain the common features all over again.)

The standalone Rabbit DSL you are to implement leaves out several features of the EDSL, including double-precision numbers, rotation, and scaling. Only the features described in this section e.g. in typing and evaluation rules need to be handled.

6.1 Compiling and Running the Standalone DSL

To compile the DSL interpreter, run `./compile.sh`. To run the DSL interpreter to produce an animation, run `./run.sh`. The script expects the first argument to be the name of a Rabbit source file, e.g. `examples/slowTurtle.rab`, and a second argument, the name of the GIF file to produce.

As an example, to run `turtleAndRabbit.rab` with the output written to `turtleAndRabbit.gif`, the command would be:

```
./run.sh examples/turtleAndRabbit.rab turtleAndRabbit.gif
```

The `run.sh` script only makes sense to run on a Rabbit program that produces an animation (that is, has type `signal[frame]`. The `test.sh` script works for any Rabbit program, and simply prints the result value as an abstract syntax tree. It takes a single argument, the name of the Rabbit program to run.

6.2 Syntax

As noted already, Rabbit's abstract syntax extends that of Giraffe. The main new features are lists and primitives that correspond to elements of the Rabbit EDSL. The syntax is summarized in Figure 7.

You do not need to write new Rabbit programs, though you may find doing so useful for testing. **If you do try to write your own Rabbit programs, be aware that the parser is a little idiosyncratic, and the error messages are very cryptic. Two specific known issues are that function application associates to the left (so you may need to add parentheses) and parsing of unary negation of integers is not supported, so to get a negative integer you need to subtract a positive integer from 0.**

In `RabbitStandalone.scala`, we define the syntax using two types, `Expr` and `Value`. Notice that list values are implemented as Scala lists, and function values do not have any type annotations. The `Value` class is defined as a subclass of `Expr`, which implies that `Value <: Expr`. In pattern matching, you can detect whether an `Expr e` is actually a `Value` using `case v: Value => ...`; on success `v` will be equal to `e` but have type `Value`. There are a few examples of this in the code already.

Values and expressions include special forms called *signal values* and *signal expressions*, respectively. Signal values are expressions built up out of the signal operations. Ordinary expressions include the signal block construct and the contents of a signal block is a signal expression. Signal expressions are similar to expressions but exclude the signal block expression itself and the various binding forms like let and case. Signal expressions also exclude the operations for working wiht pairs, lists and the unit type. Signal expressions include an *escape* expression that allows incorporating an ordinary expression (which will need to have a signal type) into a signal expression. This can be used to embed a signal already defined outside the signal expression, to allow for decomposition or reuse, for example:

$$\mathsf{let}\ turtle = \mathsf{read}\ \text{"}turtle\text{"}\ \mathsf{in}\ \mathsf{signal}\ \{\%(turtle)\ \mathsf{<+>}\ \mathsf{moveXY}(time, 100, \%(turtle))\}$$

Finally, types are considered *simple* when they contain no occurrences of the `signal`$[-]$ type constructor; and types τ occurring inside `signal`$[\tau]$ are expected to be simple, that is nesting of the `signal`$[-]$ type constructor is not allowed.

There are some example programs in the concrete syntax in the `examples` directory. For example:

```
1 let rec overAll(l : list[signal[frame]]) : signal[frame] =
2     case l { [] => blank | y :: ys =>  overAll(ys) <+> y }
3 in
4 let l =  signal { moveXY(20*time, 0 - 300, read("turtle"))} ::
5          signal { moveXY(30*time, 300, read("rabbit"))} :: ([]:signal[frame]) in
6 overAll(l)
```

defines a recursive function that superimposes all of the animations in a list, and then applies it to the turtle and rabbit animations.

6.3 Typing Rules

The typing rules for arithmetic, booleans, strings, conditionals, functions, pairs, and lists (Figure 8) are largely the same as covered in lectures/tutorials, but we have added arithmetic operations ($-$, /) and comparisons ($<, >$). Many of these can be implemented just as in Assignment 2, and implementing the rules for lists should follow a similar pattern. (Notice however that in Rabbit, empty lists are tagged with their type, which simplifies typechecking somewhat compared to the rules presented in Tutorial 3). It is safe to assume that there are no occurrences of value forms in an expression when it is being typechecked.

Several of the typechecking rules have preconditions of the form $simple(\tau)$. These conditions correspond to checking that the type τ is actually a simple type, that is, contains no occurrences of `signal`$[-]$ in it, as described in Figure 7. It is simplicit in the notation `signal`$[\tau]$ that this is the case, but in an implementation this needs to be checked in several places. The rules where this needs to be checked explicitly are those with the $simple(\tau)$ precondition. It is suggested in the implementation starter code to define a helper function `isSimple` that performs this check.

The typechecking rules are divided into two groups, one for ordinary expressions e shown in Figure 8. Many cases are just as in Giraffe; the main new cases are those for lists and for the signal expressions.

Exercise 2. *Implement the typechecking function for ordinary expressions::*

```
def tyOf(ctx: Env[Type], e: Expr): Type
```

For this exercise you do not need to handle the signal $\{e\}$ *expression. You may reuse cases from the typechecker for Giraffe from Assignment 2 (where appropriate).*

[10 marks]

Values	v	::=	$n \in \mathbb{N} \mid b \in \mathbb{B} \mid s \in \mathit{String}$	Constants
		|	$x \mid \backslash x.e \mid \mathsf{rec}\, f(x).x$	Anonymous functions
		|	$[v_1, \ldots, v_n]$	Lists
		|	(v_1, v_2)	Pairs
		|	$()$	Unit value
		|	sv	Signal value
Signal values	sv	::=	$\mathsf{pure}\, v \mid sv \mathbin{<\!\!*\!\!>} sv \mid \mathsf{time} \mid \mathsf{moveXY}(sv_1, sv_2, sv_3) \mid \mathsf{read}\, v$	
		|	$\mathsf{when}(sv_1, sv_2, sv_3) \mid \mathsf{blank} \mid sv_1 \mathbin{<\!+\!>} sv_2$	
Expressions	e	::=	$n \in \mathbb{N} \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2$	Numbers
		|	$b \in \mathbb{B} \mid \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \mid e_1 == e_2 \mid e_1 < e_2 \mid e_1 > e_2$	Booleans
		|	$s \in \mathit{String}$	Strings
		|	$x \mid \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$	Binding
		|	$\backslash x : \sigma.e \mid \mathsf{rec}\, f(x : \sigma_1) : \sigma_2.e \mid e_1\, e_2$	Functions
		|	$[] : \tau \mid e_1 :: e_2 \mid \mathsf{case}\, e \{ [\,] \Rightarrow e_1 \mid x :: y \Rightarrow e_2 \}$	Lists
		|	$(e_1, e_2) \mid \mathsf{fst}(e) \mid \mathsf{snd}(e)$	Pairs
		|	$() \mid e_1; e_2$	Unit, Sequencing
		|	$\mathsf{let}\ \mathsf{fun}\, f(x : \tau) = e_1\, \mathsf{in}\, e_2 \mid \mathsf{let}\ \mathsf{rec}\, f(x : \tau_1) : \tau_2 = e_1\, \mathsf{in}\, e_2$	
		|	$\mathsf{let}\, (x, y) = e_1\, \mathsf{in}\, e_2$	Syntactic Sugar
		|	$\mathsf{pure}\, e \mid e \mathbin{<\!\!*\!\!>} e \mid \mathsf{time} \mid \mathsf{moveXY}(e_1, e_2, e_3) \mid \mathsf{read}\, e$	
		|	$\mathsf{when}(e_1, e_2, e_3) \mid \mathsf{blank} \mid se_1 \mathbin{<\!+\!>} se_2$	Signals
		|	$\mathsf{signal}\, \{se\}$	Signal Expr's
Signal Expr's	se	::=	$n \in \mathbb{N} \mid se_1 + se_2 \mid se_1 - se_2 \mid se_1 * se_2 \mid se_1 / se_2$	Numbers
		|	$b \in \mathbb{B} \mid \mathsf{if}\, se\, \mathsf{then}\, se_1\, \mathsf{else}\, se_2 \mid se_1 == se_2 \mid se_1 < se_2 \mid se_1 > se_2$	Booleans
		|	$s \in \mathit{String}$	Strings
		|	$se_1\, se_2$	Function Application
		|	x	Variables
		|	$\mathsf{time} \mid \mathsf{moveXY}(se_1, se_2, se_3) \mid \mathsf{read}\, e$	
		|	$\mathsf{when}(se_1, se_2, se_3) \mid \mathsf{blank} \mid se_1 \mathbin{<\!+\!>} se_2$	Signals
		|	$\%(e)$	Escaping
Simple Types	τ	::=	$\texttt{int} \mid \texttt{bool} \mid \texttt{string} \mid \texttt{unit} \mid \texttt{frame} \mid \tau_1 \texttt{ -> } \tau_2 \mid \tau_1 * \tau_2 \mid \texttt{list}[\tau]$	
Types	σ	::=	$\tau \mid \texttt{signal}[\tau] \mid \sigma_1 \texttt{ -> } \sigma_2 \mid \sigma_1 * \sigma_2 \mid \texttt{list}[\sigma]$	

Figure 7: Syntax of $\lambda_{\mathsf{RABBIT}}$

The second group of typechecking rules are for signal expressions *se*, and are shown in Figure 9. Again most of the rules are straightforward. Signal expressions omit any binding forms and in many cases the rules are similar to but simpler than the corresponding ordinary expressions. However, note that there are several rules where a subexpression is an ordinary one (not a signal expression) and therefore needs to be typechecked using `tyOf`.

Exercise 3. *Implement the typechecking function for ordinary expressions::*

```
def tyOfSignal(ctx: Env[Type], e: Expr): Type
```

as well as the missing case of signal $\{-\}$ *in* `tyOf`.

[5 marks]

$\boxed{\Gamma \vdash e : \sigma}$

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 - e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 * e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 / e_2 : \texttt{int}}$$

$$\frac{}{\Gamma \vdash b : \texttt{bool}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 == e_2 : \texttt{bool}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 < e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 > e_2 : \texttt{bool}} \qquad \frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash e_1 : \sigma \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \sigma}$$

$$\frac{}{\Gamma \vdash s : \mathit{String}} \qquad \frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \sigma_2}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \texttt{->} \sigma_2 \qquad \Gamma \vdash e_2 : \sigma_1}{\Gamma \vdash e_1\, e_2 : \sigma_2} \qquad \frac{\Gamma, x : \sigma_1 \vdash e : \sigma_2}{\Gamma \vdash \backslash x : \sigma_1 . e : \sigma_1 \texttt{->} \sigma_2} \qquad \frac{\Gamma, f : \sigma_1 \texttt{->} \sigma_2, x : \sigma_1 \vdash e : \sigma_2}{\Gamma \vdash \mathsf{rec}\ f(x : \sigma_1) : \sigma_2 . e : \sigma_1 \texttt{->} \sigma_2}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash (e_1, e_2) : \sigma_1 * \sigma_2} \qquad \frac{\Gamma \vdash e : \sigma_1 * \sigma_2}{\Gamma \vdash \mathsf{fst}(e) : \sigma_1} \qquad \frac{\Gamma \vdash e : \sigma_1 * \sigma_2}{\Gamma \vdash \mathsf{snd}(e) : \sigma_2}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 * \sigma_2 \qquad \Gamma, x : \sigma_1, y : \sigma_2 \vdash e_2 : \sigma}{\Gamma \vdash \mathsf{let}\ (x, y) = e_1\ \mathsf{in}\ e_2 : \sigma}$$

$$\frac{}{\Gamma \vdash [\,] : \sigma : \texttt{list}[\sigma]} \qquad \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma \vdash e_2 : \texttt{list}[\sigma]}{\Gamma \vdash e_1 :: e_2 : \texttt{list}[\sigma]}$$

$$\frac{\Gamma \vdash e : \texttt{list}[\sigma_1] \qquad \Gamma \vdash e_1 : \sigma_2 \qquad \Gamma, x : \sigma_1, y : \texttt{list}[\sigma_1] \vdash e_2 : \sigma_2}{\Gamma \vdash \mathsf{case}\ e \{ [\,] \texttt{=>} e_1 \mid x :: y \texttt{=>} e_2 \} : \sigma_2} \qquad \frac{}{\Gamma \vdash () : \texttt{unit}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{unit} \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 ; e_2 : \sigma} \qquad \frac{\Gamma \vdash e : \tau \qquad \mathit{simple}(\tau)}{\Gamma \vdash \mathsf{pure}\ e : \texttt{signal}[\tau]}$$

$$\frac{\Gamma \vdash e_1 : \texttt{signal}[\tau_1 \texttt{->} \tau_2] \qquad \Gamma \vdash e_2 : \texttt{signal}[\tau_1]}{\Gamma \vdash e_1 \texttt{<*>}\ e_2 : \texttt{signal}[\tau_2]} \qquad \frac{}{\Gamma \vdash \mathsf{time} : \texttt{signal}[\texttt{int}]}$$

$$\frac{}{\Gamma \vdash \mathsf{blank} : \texttt{signal}[\texttt{frame}]} \qquad \frac{\Gamma \vdash e_1 : \texttt{signal}[\texttt{frame}] \qquad \Gamma \vdash e_2 : \texttt{signal}[\texttt{frame}]}{\Gamma \vdash e_1 \texttt{<+>}\ e_2 : \texttt{signal}[\texttt{frame}]}$$

$$\frac{\Gamma \vdash e_1 : \texttt{signal}[\texttt{int}] \qquad \Gamma \vdash e_2 : \texttt{signal}[\texttt{int}] \qquad \Gamma \vdash e_1 : \texttt{signal}[\texttt{frame}]}{\Gamma \vdash \mathsf{moveXY}(e_1, e_2, e_3) : \texttt{signal}[\texttt{frame}]}$$

$$\frac{\Gamma \vdash e_1 : \texttt{signal}[\texttt{bool}] \qquad \Gamma \vdash e_2 : \texttt{signal}[\tau] \qquad \Gamma \vdash e_1 : \texttt{signal}[\tau]}{\Gamma \vdash \mathsf{when}(e_1, e_2, e_3) : \texttt{signal}[\tau]} \qquad \frac{\Gamma \vdash se :_{\mathit{sig}} \tau}{\Gamma \vdash \mathsf{signal}\ \{se\} : \texttt{signal}[\tau]}$$

$$\frac{\Gamma \vdash e : \texttt{string}}{\Gamma \vdash \mathsf{read}\ e : \texttt{signal}[\texttt{string}]}$$

Figure 8: Typing rules for ordinary expressions

$\boxed{\Gamma \vdash se :_{sig} \tau}$

$$\frac{}{\Gamma \vdash n :_{sig} \texttt{int}} \qquad \frac{\Gamma \vdash se_1 :_{sig} \texttt{int} \qquad \Gamma \vdash se_2 :_{sig} \texttt{int}}{\Gamma \vdash se_1 + se_2 :_{sig} \texttt{int}} \qquad \frac{\Gamma \vdash se_1 :_{sig} \texttt{int} \qquad \Gamma \vdash se_2 :_{sig} \texttt{int}}{\Gamma \vdash se_1 - se_2 :_{sig} \texttt{int}}$$

$$\frac{\Gamma \vdash se_1 :_{sig} \texttt{int} \qquad \Gamma \vdash se_2 :_{sig} \texttt{int}}{\Gamma \vdash se_1 * se_2 :_{sig} \texttt{int}} \qquad \frac{\Gamma \vdash se_1 :_{sig} \texttt{int} \qquad \Gamma \vdash se_2 :_{sig} \texttt{int}}{\Gamma \vdash se_1 / se_2 :_{sig} \texttt{int}}$$

$$\frac{}{\Gamma \vdash b :_{sig} \texttt{bool}} \qquad \frac{\Gamma \vdash se_1 :_{sig} \tau \qquad \Gamma \vdash se_2 :_{sig} \tau \qquad \tau \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash se_1 == se_2 :_{sig} \texttt{bool}}$$

$$\frac{\Gamma \vdash se_1 :_{sig} \texttt{int} \qquad \Gamma \vdash se_2 :_{sig} \texttt{int}}{\Gamma \vdash se_1 < se_2 :_{sig} \texttt{bool}} \qquad \frac{\Gamma \vdash se_1 :_{sig} \texttt{int} \qquad \Gamma \vdash se_2 :_{sig} \texttt{int}}{\Gamma \vdash se_1 > se_2 :_{sig} \texttt{bool}}$$

$$\frac{\Gamma \vdash se :_{sig} \texttt{bool} \qquad \Gamma \vdash se_1 :_{sig} \tau \qquad \Gamma \vdash se_2 :_{sig} \tau}{\Gamma \vdash \mathsf{if}\ se\ \mathsf{then}\ se_1\ \mathsf{else}\ se_2 :_{sig} \tau}$$

$$\frac{}{\Gamma \vdash s :_{sig} \mathit{String}} \qquad \frac{\Gamma(x) = \tau \qquad \mathit{simple}(\tau)}{\Gamma \vdash x :_{sig} \tau} \qquad \frac{\Gamma \vdash se_1 :_{sig} \tau_1 \texttt{->} \tau_2 \qquad \Gamma \vdash se_2 :_{sig} \tau_1}{\Gamma \vdash se_1\ se_2 :_{sig} \tau_2}$$

$$\frac{}{\Gamma \vdash \mathsf{time} :_{sig} \texttt{int}} \qquad \frac{}{\Gamma \vdash \mathsf{blank} :_{sig} \texttt{frame}} \qquad \frac{\Gamma \vdash se_1 :_{sig} \texttt{frame} \qquad \Gamma \vdash se_2 :_{sig} \texttt{frame}}{\Gamma \vdash se_1 \texttt{<+>} se_2 :_{sig} \texttt{frame}}$$

$$\frac{\Gamma \vdash se_1 :_{sig} \texttt{int} \qquad \Gamma \vdash se_2 :_{sig} \texttt{int} \qquad \Gamma \vdash se_3 :_{sig} \texttt{frame}}{\Gamma \vdash \mathsf{moveXY}(se_1, se_2, se_3) :_{sig} \texttt{frame}}$$

$$\frac{\Gamma \vdash se_1 :_{sig} \texttt{bool} \qquad \Gamma \vdash se_2 :_{sig} \tau \qquad \Gamma \vdash se_3 :_{sig} \tau}{\Gamma \vdash \mathsf{when}(se_1, se_2, se_3) :_{sig} \tau} \qquad \frac{\Gamma \vdash e : \texttt{string}}{\Gamma \vdash \mathsf{read}\ e :_{sig} \texttt{string}} \qquad \frac{\Gamma \vdash e : \mathsf{signal}\ \{\tau\}}{\Gamma \vdash \%(e) :_{sig} \tau}$$

Figure 9: Typing rules for signal expressions

6.4 Substitution and Desugaring

Capture-avoiding substitution is needed for $\lambda_{\mathsf{RABBIT}}$ both for desugaring and for evaluation, since the semantics you are to implement will use substitution to replace variables with their values. Luckily, we have defined the swapping operation for you, and most of the cases of substitution are similar to those for the Giraffe language of Assignment 2, so you can reuse them. You only need to fill in the new cases, such as for lists and the Rabbit signal primitives.

Function values may have expressions as subterms, but we expect that values are always *closed*, that is, have no free variables. Thus, for the case when the expression is actually a value `v: Value`, substitution does not need to do any work (this case is already done in the starting code).

Exercise 4. *Define the substitution function* `subst` *for* $\lambda_{\mathsf{RABBIT}}$ *expressions. (You may copy over the sample solution from Assignment 2 to use as a starting point.)*

[10 marks]

Several constructs of $\lambda_{\mathsf{RABBIT}}$ are definable using others. We already saw examples of this in Assignment 2, such as `let fun`, `let rec`, `let pair`. Among the constructs of $\lambda_{\mathsf{RABBIT}}$, the following can be defined in terms of others using simple rewriting rules:

$$
\begin{array}{rcl}
\mathsf{let\ fun}\ f(x:\tau) = e_1\ \mathsf{in}\ e_2 & \iff & \mathsf{let}\ f = \backslash x:\tau.e_1\ \mathsf{in}\ e_2 \\
\mathsf{let\ rec}\ f(x:\tau_1):\tau_2 = e_1\ \mathsf{in}\ e_2 & \iff & \mathsf{let}\ f = \mathsf{rec}\ f(x:\tau_1):\tau_2.e_1\ \mathsf{in}\ e_2 \\
\mathsf{let}\ (x,y) = e_1\ \mathsf{in}\ e_2 & \iff & \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2[\mathsf{fst}(p)/x][\mathsf{snd}(p)/y] \\
e_1; e_2 & \iff & \mathsf{let}\ x = e_1 = e_2\ \mathsf{in} \qquad (x \notin FV(e_2)
\end{array}
$$

When desugaring, make sure to replace any syntactic sugar inside values, by handling the cases for `FunV(x,e)` and `RecV(f,x,e)`.

Exercise 5. *Implement the function* `desugar` *that replaces the above constructs in* $\lambda_{\mathsf{RABBIT}}$ *with their desugared forms. (As before, you may use the similar cases for Giraffe as a starting point.) You do not need to implement the case for* signal $\{se\}$ *which is the subject of the next exercise.*

[5 marks]

Next we consider desugaring the signal blocks. The signal block operations desugar as follows:

$$
\begin{array}{rcl}
\mathsf{signal}\ \{se\} & \iff & dsBlock(se) \\
dsBlock(v) & = & \mathsf{pure}\ v \\
dsBlock(se_1\ se_2) & = & dsBlock(se_1)\ \mathsf{<*>}\ dsBlock(se_2) \\
dsBlock(\mathsf{if}\ se\ \mathsf{then}\ se_1\ \mathsf{else}\ se_2) & = & \mathsf{when}(dsBlock(se), dsBlock(se_1), dsBlock(se_2)) \\
dsBlock(se_1\ \oplus\ se_2) & = & \mathsf{pure}\ (\lambda x : \mathtt{int}.\lambda y : \mathtt{int}.x \oplus y)\ \mathsf{<*>}\ dsBlock(se_1)\ \mathsf{<*>}\ dsBlock(se_2) \\
dsBlock(\mathsf{time}) & = & \mathsf{time} \\
dsBlock(\mathsf{read}\ (e)) & = & \mathsf{read}\ (\mathtt{desugar}(e)) \\
dsBlock(\mathsf{moveXY}(se_1, se_2, se_3)) & = & \mathsf{moveXY}(dsBlock(se_1), dsBlock(se_2), dsBlock(se_3)) \\
dsBlock(\%(e)) & = & \mathtt{desugar}(e)
\end{array}
$$

Note again that in implementing the *ds* function, there are two cases where you will need to call the `desugar` function to desugar ordinary expressions that are embedded in signal expressions. The case for $\oplus$ summarizes how to treat the various binary operations and relations, all of which are restricted to apply to integer inputs in the standalone language. It may be helpful to create a helper function that handles these cases.

Exercise 6. *Implement the function* `desugarBlock` *as well as the case of* signal $\{se\}$ *in* `desugar` *that should call it.*

[10 marks]

6.5 Evaluation

The semantics of λ_{RABBIT} is defined using large-step evaluation rules. The evaluation judgment is as follows:

$$e \Downarrow v$$

where e is the expression to be evaluated, and v is the value.

Most rules are similar to those given in lectures, specifically for arithmetic, booleans, variables, let-binding, functions, pairs and lists, as shown in Figure 10. The main novelty in this evaluator is that signals are evaluated to values with complex structure: the *signal values* shown in Figure 7. In the implementation, signal values are represented as elements of the `RabbitEDSL.Signal[T]` type. Thus when implementing these cases it is sufficient to evaluate the subexpressions to their values and (for signal-valiued subexpressions) extract the `RabbitEDSL.Signal[T]` values and use the corresponding `RabbitEDSL` operation to construct the appropriate result type.

No rules are given for the signal block expression signal $\{se\}$ or for the signal expression cases. These cases are eliminated through the desugaring of signal blocks performed by `desugar` and `desugarSignal`.

6.6 Evaluation Model

A Rabbit program that has type `signal[frame]` is evaluated, hopefully resulting in a signal value which is essentially the same as an expression tree for the Rabbit EDSL. However, it is not straightforward todirectly evaluate this expression directly using the `RabbitDSL` interface, essentially because it is difficult to write a function from `Values` to `RabbitDSL.RabbitAnimation[Frame]` because of the need to handle cases where signals of other types besides `Frame` are encountered.

Instead we follow a *staged* approach: once we have evaluated a Rabbit expression of type `signal[frame]` to a signal value, we traverse it and generate equivalent *Scala code* using the `RabbitDSL` operations, save this code to a file and execute it. This is what `run.sh` does.

6.7 Implementing the Evaluator

The `Eval` class provides a function `run` to evaluate whole λ_{RABBIT} programs. The `run` method is located in a class `Eval` that takes an instance of the `RabbitEDSL` trait as a parameter, and import it, so that we can run λ_{RABBIT} using different instances of the `RabbitEDSL` trait, such as `Testing` for testing or `RabbitEDSLImpl` to actually create the animation. The `run` function is given an expression and output filename, and does the bureaucratic work of setting up the graphics canvas and other setup. It calls a (private) helper function *eval* that takes an expression and yields a value. The value is then embedded in a temporary Scala file which is run in order to produce the desired animation. The `eval` method should follow the inference rules given in Figure 10.

Exercise 7. *Implement the evaluator* `eval`*, whose type signature is as follows:*

```
def eval(expr: Expr): Value
```

[20 marks]

$\boxed{\sigma, e \Downarrow v}$

$$\frac{v \text{ is a value}}{v \Downarrow v} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 + e_2 \Downarrow v_1 +_{\mathbb{N}} v_2} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 - e_2 \Downarrow v_1 -_{\mathbb{N}} v_2} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 * e_2 \Downarrow v_1 *_{\mathbb{N}} v_2}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 / e_2 \Downarrow v_1 /_{\mathbb{N}} v_2}$$

$$\frac{e \Downarrow \mathsf{true} \qquad e_1 \Downarrow v}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v} \qquad \frac{e \Downarrow \mathsf{false} \qquad e_2 \Downarrow v}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v} \qquad \frac{e_1 \Downarrow v \qquad e_2 \Downarrow v}{e_1 == e_2 \Downarrow \mathsf{true}}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 \neq v_2}{e_1 == e_2 \Downarrow \mathsf{false}} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 < e_2 \Downarrow v_1 <_{\mathbb{N}} v_2} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 > e_2 \Downarrow v_1 >_{\mathbb{N}} v_2}$$

$$\frac{e_1 \Downarrow \lambda x.e \qquad e_2 \Downarrow v_1 \qquad e[v_1/x] \Downarrow v_2}{e_1\, e_2 \Downarrow v_2}$$

$$\frac{e_1 \Downarrow \mathsf{rec}\, f(x:\tau):\tau'.e \qquad e_2 \Downarrow v_1 \qquad e[v_1/x, \mathsf{rec}\, f(x:\tau):\tau'.e/f] \Downarrow v_2}{e_1\, e_2 \Downarrow v_2}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2[v_1/x] \Downarrow v}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow v}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 :: e_2 \Downarrow v_1 :: v_2} \qquad \frac{e \Downarrow [\,] : \tau \qquad e_1 \Downarrow v_1}{\mathsf{case}\, e \{ [\,] => e_1 \mid x :: y => e_2 \} \Downarrow v_1} \qquad \frac{e \Downarrow v_1 :: v_2 \qquad e_2[v_1/x, v_2/y] \Downarrow v}{\mathsf{case}\, e \{ [\,] => e_1 \mid x :: y => e_2 \} \Downarrow v}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(e_1, e_2) \Downarrow (v_1, v_2)} \qquad \frac{e \Downarrow (v_1, v_2)}{\mathsf{fst}(e) \Downarrow v_1} \qquad \frac{e \Downarrow (v_1, v_2)}{\mathsf{snd}(e) \Downarrow v_1} \qquad \frac{e \Downarrow v}{\mathsf{pure}\ e \Downarrow \mathsf{pure}\ v}$$

$$\frac{e_1 \Downarrow sv_1 \qquad e_2 \Downarrow sv_2}{e_1 <\!\!*\!\!> e_2 \Downarrow sv_1 <\!\!*\!\!> sv_2} \qquad \frac{e_1 \Downarrow sv_1 \qquad e_2 \Downarrow sv_2 \qquad e_3 \Downarrow sv_3}{\mathsf{moveXY}(e_1, e_2, e_3) \Downarrow \mathsf{moveXY}(sv_1, sv_2, sv_3)}$$

$$\frac{e_1 \Downarrow sv_1 \qquad e_2 \Downarrow sv_2 \qquad e_3 \Downarrow sv_3}{\mathsf{when}(e_1, e_2, e_3) \Downarrow \mathsf{when}(sv_1, sv_2, sv_3)} \qquad \frac{e \Downarrow v}{\mathsf{read}\ e \Downarrow \mathsf{read}\ v} \qquad \frac{e_1 \Downarrow sv_1 \qquad e_2 \Downarrow sv_2}{e_1 <\!\!+\!\!> e_2 \Downarrow sv_1 <\!\!+\!\!> sv_2}$$

Figure 10: Evaluation rules

Change Log

- v1.1: Added to and changed the instructions for running and using scripts, including ability to run using a sample solution (section 4 and references to `sbt compile` in sections 5 and 6).