# Elements of Programming Languages
# Assignment 3: Turtle Graphics
# Version 1.5 (last updated November 23)
# Due: November 23, 4pm

November 23, 2016

## 1 Introduction

Turtle Graphics[1] is an educational system used to teach introductory programming. In Turtle Graphics systems, a turtle can be moved around a canvas given commands written in some *domain-specific language* (generally a dialect of LOGO).

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].

In this assignment, we will investigate two ways of implementing a domain specific language for Turtle Graphics. 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, $\lambda_{\mathsf{LOGO}}$. 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 4pm.

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 Turtle Graphics

The idea behind Turtle Graphics is to move a turtle around the screen, which may draw a line as it moves between two points.

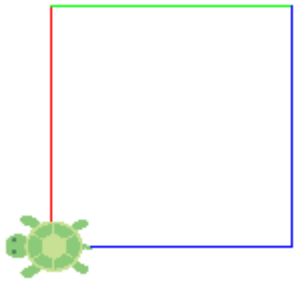
Figure 1 shows a simple $\lambda_{\mathsf{LOGO}}$ program to draw a square. The setCol construct sets the colour of the line to be drawn by the turtle; forward moves the turtle forwards by a given number of units, and right rotates the turtle by a given number of degrees.

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

[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  setCol Red;
2  forward 120;
3  right 90;
4  setCol Green;
5  forward 120;
6  right 90;
7  setCol Blue;
8  forward 120;
9  right 90;
10 forward 120
```

(a) $\lambda_{\text{LOGO}}$ program

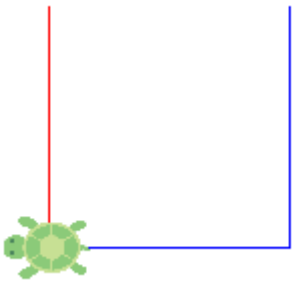


(b) Output Image

Figure 1: Drawing a Square with $\lambda_{\text{LOGO}}$

```
1  setCol Red;
2  forward 120;
3  right 90;
4  penUp;
5  setCol Green;
6  forward 120;
7  penDown;
8  right 90;
9  setCol Blue;
10 forward 120;
11 right 90;
12 forward 120
```

(a) $\lambda_{\text{LOGO}}$ program



(b) Output Image

Figure 2: Drawing a Square with $\lambda_{\text{LOGO}}$, with the pen up for one side

It may be the case that we wish to move the turtle *without* drawing a line. To handle this, we have two constructs `penUp` and `penDown`. Intuitively, if the pen is down, then a line will be drawn whenever the turtle moves, whereas if the pen is up, the turtle will move but will not draw a line. Figure 2 shows a similar program, but with the pen up when drawing the top side of the square.

Of course, this alone is rather uninteresting! Surprisingly, given variables, recursion, and looping, we can quickly make some more interesting graphics. Figure 3 shows a simple fractal, where squares are drawn recursively within squares. 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 $\lambda_{\text{LOGO}}$ programs of your own.

Concretely, the turtle-specific constructs are as follows:

**penUp** lifts the pen so that drawing no longer takes place.

**penDown** lowers the pen so that drawing takes place.

**forward** $n$ moves the turtle forward by $n$ units.

**backward** $n$ moves the turtle forward by $n$ units.

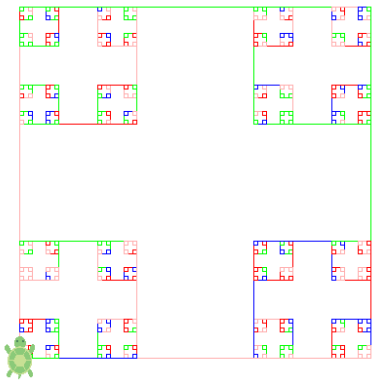
**left** $n$ rotates the turtle anticlockwise by $n$ degrees.

**right** $n$ rotates the turtle clockwise by $n$ degrees.

**setCol** $c$ sets the colour of the lines that will be drawn by the turtle, where $c$ is one of Red, Green, Blue, or Pink.

```
let colList = Red::Green::Blue::Pink::([]: color) in
let rec drawFractal(p: (int * int)): unit =
  let (length, depth) = p in
  if (depth == 0) then ()
  else
    let counterRef = ref 0 in
    randCol colList;
    while (!counterRef < 4) {
      forward length;
      right 90;
      drawFractal((length / 3), (depth - 1));
      counterRef := (!counterRef + 1)
    } in
drawFractal((360, 5))
```

(a) $\lambda_{\text{LOGO}}$ program



(b) Output Image

Figure 3: Drawing a Square Fractal with $\lambda_{\text{LOGO}}$

**randCol** *cs* sets the colour of the lines drawn by the turtle to a colour randomly selected from a list *cs*.

# 3 Included Code: a Simple Graphics Library

```
trait GraphicsCanvasTrait {
  def drawLine(x0: Integer, y0: Integer, x1: Integer, y1: Integer): Unit
  def setLineColor(col: java.awt.Color): Unit
  def drawTurtle(x: Integer, y: Integer, angle: Integer): Unit
  def saveToFile(filename: String): Unit
}
```

To help you get started, we provide `GraphicsCanvas.scala`, which is a simple graphics library to wrap around Java's `Graphics2D` library. You should not need to use any of the Java graphics libraries directly. The operations are as follows:

- `drawLine(x0: Integer, y0: Integer, x1: Integer, y1: Integer): Unit` draws a line from the point with co-ordinates $(x0, y0)$ to the point with co-ordinates $(x1, y1)$.
- `setLineColor(col: java.awt.Color)` sets the colour of subsequent lines that are drawn to the canvas.
- `drawTurtle(x: Integer, y: Integer, angle: Integer): Unit` draws a turtle at co-ordinates $(x, y)$, rotated by `angle` degrees clockwise.
- `saveToFile(filename: String): Unit` saves the contents of the canvas as a PNG file, at a path given by `filename`.

## 3.1 Co-ordinate System

The co-ordinate system begins with $(0, 0)$ in the *top-left corner* of the canvas. Co-ordinates are integer values, where each integer refers to a pixel.

Figure 4 shows example co-ordinates for a canvas of size $100 \times 200$. Note that negative co-ordinates are not valid. The turtle should start in the middle of the canvas (the point (50,100) in this example).

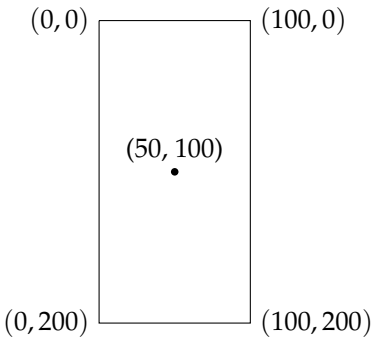
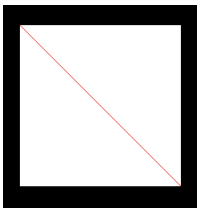



Figure 4: Co-ordinate System

```
1 import Assignment3.GraphicsCanvas._
2 val w = 200
3 val h = 200
4 val canvas = new GraphicsCanvas(w, h)
5 canvas.setLineColor(java.awt.Color.RED)
6 canvas.drawLine(0, 0, w, h)
7 canvas.saveToFile("diag.png")
```

(a) Code to draw a diagonal line on a canvas



(b) Resulting image

Figure 5: Drawing on a GraphicsCanvas

### 3.2 Basic Usage

To create a new `GraphicsCanvas` instance, simply call the constructor with two parameters stating the width and height of the canvas. As an example, the following code:

- Creates a new 200 × 200px canvas (lines 2-4)
- Sets the colour of future lines to red (line 5)
- Draws a diagonal line from the top-left corner to the bottom-right corner (line 6)
- Saves the result to the file `diag.png` (line 7)

Note that the operations are all side-effecting and return the unit value.

## 4 Getting started

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

- `GraphicsCanvas.scala`, giving an implementation of the graphics canvas object. You should not need to change this file.
- `TurtleEDSL.scala`, containing the supporting code and template for the Turtle EDSL.
- `TurtleStandalone.scala`, containing the supporting code and template for the standalone implementation of $\lambda_{\mathsf{LOGO}}$.

We include several example programs. Some are defined using the EDSL approach in `TurtleEDSL.scala`, and `run_edsl.sh` will run these examples and generate PNG output. Others are provided as files for use with the standalone DSL, and can be run using `run_standalone.sh`.

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

- `compile_edsl.sh` compiles the EDSL code

- `compile_standalone.sh` compiles the standalone code
- `compile.sh` compiles everything
- `run_edsl.sh` runs the EDSL code
- `run_standalone.sh` runs the standalone interpreter (on a file)

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.

Finally, we provide two JAR files that contain a sample solutions called `TurtleEDSL.jar` and `TurtleStandalone.jar`. You can run them as follows:

```
$ scala TurtleEDSL.jar                 # runs the EDSL on some examples
$ scala TurtleStandalone.jar <filename> # runs the interpreter on a file
```

### 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. Some specific pointers to library primitives such as for random number generation are mentioned later.

**This assignment relies on material covered up to Lecture 14 (November 15). The two parts of this assignment are independent and can be attempted in either order.**

This assignment is graded on a scale of 25 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. To submit, use the following DICE commands:

```
$ zip Assn3.zip TurtleEDSL.scala TurtleStandalone.scala
$ submit epl 3 Assn3.zip
```

The submission deadline is 4pm on November 23.

## 5 Part 1: An Embedded Domain-Specific Language

In this section, you will implement Turtle Graphics 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 `TurtleEDSL.scala`.

## 5.1 Compiling and Running the EDSL

**Compiling the DSL** The `./compile_edsl.sh` script will compile the DSL, as well as `GraphicsCanvas` if required.

**Running the DSL** The `TurtleEDSL.scala` file contains a definition `val toRun = List(...)`, containing a list of (`TurtleGraphics`, `Filename`) pairs. To run the file, run `./run_edsl.sh`, or alternatively:

```
scala -cp . Assignment3.TurtleEDSL.Assignment3Embedded
```

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

If you wish to test the functions individually from the prompt, you can do as follows:

```
scala -cp .
scala> :load TurtleEDSL.scala
scala> import Assignment3Embedded._
```

Note that you might get an `illegal start of definition` error when Scala comes across the package name: this can safely be ignored.

The `TurtleEDSL` file includes an object `Testing` that "implements" `TurtleDSL` 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 TurtleDSLImpl._
```

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 TurtleDSL {
2    type TurtleGraphics
3    val empty: TurtleGraphics
4    def append(tg1: TurtleGraphics, tg2: TurtleGraphics): TurtleGraphics
5    def penUp(): TurtleGraphics
6    def penDown(): TurtleGraphics
7    def forward(distance: Integer): TurtleGraphics
8    def backward(distance: Integer): TurtleGraphics
9    def right(amount: Integer): TurtleGraphics
10   def left(amount: Integer): TurtleGraphics
11   def setColor(color: Color): TurtleGraphics
12   def setRandomColor(cols: List[Color]): TurtleGraphics
13   def calculateAngleDiff(angle: Integer, diff: Integer): Integer
14   def calculateNewCoords(x0: Integer, y0: Integer, angle: Integer,
15     distance: Integer): (Integer, Integer)
16   def draw(tg: TurtleGraphics, width: Integer,
17     height: Integer): GraphicsCanvas
18   def saveToFile(tg: TurtleGraphics, width: Integer,
19     height: Integer, filename: String): Unit
20 }
```

Here, `TurtleGraphics` is an abstract type which you will have to define in your implementation. The constructs are as follows:

- `empty: TurtleGraphics` an operation which does nothing.

- `append(tg1: TurtleGraphics, tg2: TurtleGraphics): TurtleGraphics` an operation which firstly performs `tg1`, and then performs `tg2`.
- `penUp(): TurtleGraphics`: an operation which raises the pen, meaning that lines will *not* be drawn when the turtle moves.
- `penUp(): TurtleGraphics`: an operation which lowers the pen, meaning that lines *will* be drawn when the turtle moves.
- `forward(distance: Integer): TurtleGraphics`: an operation which moves the turtle forward by `distance` pixels
- `backward(distance: Integer): TurtleGraphics`: an operation which moves the turtle backward by `distance` pixels
- `right(amount: Integer): TurtleGraphics`: an operation which rotates the turtle right by `amount` degrees. Note that the angle is calculated modulo 360 deg, so `right(5)` with a current angle of 359 will result in an angle of 4.
- `left(amount: Integer): TurtleGraphics`: an operation which rotates the turtle left by `amount` degrees. Again, the angle is calculated modulo 360 deg, so `left(5)` with a current angle of 4 will result in an angle of 359.
- `setColor(color: java.awt.Color): TurtleGraphics`: an operation which sets the colour of subsequent lines to be `color`.
- `setRandomColor(cols: List[java.awt.Color]): TurtleGraphics`: an operation which sets the current line colour to a random colour from `cols`. Fails using `sys.error` if `cols` is empty.
- `def calculateAngleDiff(angle: Integer, diff: Integer): Integer`: Calculates an angle difference modulo 360. So: `calculateAngleDiff`$(359, 5) = 4$, and `calculateAngleDiff`$(4, -5) = 359$.
- `calculateNewCoords(x0: Integer, y0: Integer, angle: Integer, distance: Integer): (Integer, Integer)`: takes the current $x$ and $y$ co-ordinates ($x_0$ and $y_0$), the current angle, and the distance to move forward. Returns a pair of the co-ordinates after the move has taken place.
- `draw(tg: TurtleGraphics, width: Integer, height:Integer): GraphicsCanvas`: takes a `TurtleGraphics` command, runs it on a canvas of width `width` px and height `height` px, and returns the resulting `GraphicsCanvas`.
- `saveToFile(tg: TurtleGraphics, width: Integer, height: Integer,filename: String)`: takes a `TurtleGraphics` command, runs it on a canvas of width `width` px and height `height` px, and saves the resulting canvas as a PNG file to the location given by `filename`, returning `Unit`.

For example, using the DSL, we can write a program to draw a square as follows:

```
1  def square() = {
2    forward(100) <> right(90) <>
3    forward(100) <> right(90) <>
4    forward(100) <> right(90) <>
5    forward(100) <> right(90)
6  }
```

Note that we can sequence operations using the <> infix operator. (There is some magic Scala code in the trait to support this, which you do not need to understand and should not change.)

The Turtle DSL trait contains a few operations that can be defined once and for all as part of the trait, either because they are definable in terms of other operations (e.g. `backward`, `left`), or because they are helper functions (e.g. `calculateAngleDiff`, `calculateNewCoords`, and `saveToFile`) that do not depend on the implementation type `TurtleGraphics`. We have filled in one example. namely `saveToFile`.

---

**Exercise 1.** *Implement the operations* `backward`, `left`, `calculateAngleDiff`, *and* `calculateNewCoords` *in the trait* `TurtleDSL`.

*[2 marks]*

Finally, you are to define an implementation of the `TurtleDSL` trait. The rest of this section gives additional suggestions regarding how to proceed.

**Exercise 2.** *Define an object* `TurtleDSLImpl` *extending* `TurtleDSL` *that provides definitions for all of the remaining components.*

*[8 marks]*

## 5.3 State and Geometry

In implementing the EDSL, you will need to keep track of several properties: in particular, the canvas to be drawn on, the current co-ordinates of the turtle, whether the pen is up or down, and the angle of the turtle. In what follows, we assume that the angles are measured clockwise relative to the y-axis, so that angle 0 corresponds to the turtle pointing "up". (This choice is somewhat arbitrary.)

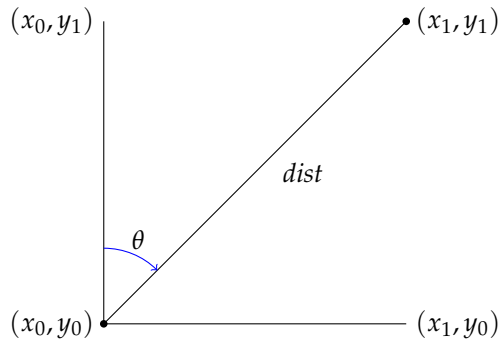



Figure 6: Calculating destination co-ordinates from source co-ordinates

You will also notice that `drawLine` requires two pairs of co-ordinates: one which will be the position before the turtle moves, and one which will be the position after the turtle moves. In order to calculate this, you will need to calculate the destination co-ordinates from the initial co-ordinates, as shown in Figure 4. Given initial co-ordinates $(x_0, y_0)$; a distance *dist*, and an angle $\theta$, the destination co-ordinates $(x_1, y_1)$ can be calculated as follows:

- $x_1 = x_0 + (dist \times sin(\theta))$
- $y_1 = y_0 - (dist \times cos(\theta))$

Important: the `math.sin(angle: Double)` and `math.cos(angle: Double)` functions take an angle in *radians*! Ensure that you convert your angle to radians using `math.toRadians(angle)` before passing them to the trigonometric functions.

## 5.4 Randomness

Scala (like Java) has a built-in random number generator library. In the provided code, the following line

```
val rand = new Random(System.currentTimeMillis())
```

initializes a random number generator called `rand`, which you should use to implement the `setRandomColor` operation. The function `rand.nextInt(n:Integer):Integer` chooses a random number between 0 and $n - 1$.

## 5.5 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 `TurtleGraphics` 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. The state is kept as a parameter to the interpreter function. An example structure is:

```
object DeepTurtleDSL extends TurtleDSL {
  type TurtleGraphics = TurtleGraphicsAST

  // AST Definition
  abstract class TurtleGraphicsAST
  case class TGEmpty() extends TurtleGraphicsAST
  case class TGForward(distance: Integer) extends TurtleGraphicsAST
  ...

  // Language constructs create AST nodes
  val empty = TGEmpty
  def forward(distance: Integer): TurtleGraphics = TGForward(distance)
  ...

  final case class TurtleGraphicsState(isPenUp: Boolean, ...)

  // draw "interprets" the AST
  def draw(prog: TurtleGraphics, width: Integer,
    height: Integer): GraphicsCanvas = {
    ...
  }
}
```

**Shallow Embeddings** A *shallow embedding* implements each construct as a function which takes the state as an argument. As language constructs can both read *and* write to the state, the function will take an input state, and produce an output state.

```
object ShallowTurtleDSL extends TurtleDSL {
  type TurtleGraphics = (TurtleGraphicsState => TurtleGraphicsState)
  final case class TurtleGraphicsState(isPenUp: Boolean, ...)

  val empty = (st: TurtleGraphicsState) => st
  def forward(distance: Integer) = (st: TurtleGraphicsState) => ...
  def draw(prog: TurtleGraphics, width: Integer,
    height: Integer): GraphicsCanvas = {
      val initialState = ...
      val finalState = prog(initialState)
      ...
  }
}
```

# 6 Part 2: A Standalone DSL

In this part of the assignment you will implement a *standalone* DSL variant of Turtle called $\lambda_{\text{LOGO}}$. "Standalone" means that we are not re-using Scala as a host language, so we need to parse in $\lambda_{\text{LOGO}}$ 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 Turtle 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.)

## 6.1 Compiling and Running the Standalone DSL

To compile the DSL interpreter, run `./compile_standalone.sh`. To run the DSL interpreter, run `./run_standalone.sh`. The script takes three optional arguments, along with the file to run. The optional arguments are:

- `-o output_filename` will save the result to `output_file`. Default: `output.png`.
- `-w width` will set the canvas width to `width`. Default: 800.
- `-h height` will set the canvas height to `height`. Default: 800.
- `-t` will run the program using the `Testing` implementation of `TurtleDSL`. You can use this to test your implementation of $\lambda_{\text{LOGO}}$ even if you don't have a working implementation of the EDSL.

As an example, to run `spiral.tg` on a canvas of size 1000 by 1500 with the output written to `spiral.png`, the command would be:

```
./run_standalone.sh -o spiral.png -w 1000 -h 1500 spiral.tg
```

## 6.2 Syntax

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

In `TurtleStandalone.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.

There are some example programs in the concrete syntax in the files called `squareFractal.tg`, `sun.tg` and so on. For example:

```
let colList = Red::Green::Blue::Pink::([] : color) in
let rec drawFractal(p: (int * int)): unit =
  let (length, depth) = p in
  if (depth == 0) then ()
  else
    let counterRef = ref 0 in
    randCol colList;
    while (!counterRef < 4) {
      forward length;
      right 90;
      drawFractal((length / 3), (depth - 1));
      counterRef := (!counterRef + 1)
    } in
drawFractal((360, 5))
```

| | | | | |
|---|---|---|---|---|
| Values | $v$ | ::= | $n \in \mathbb{N} \mid b \in \mathbb{B}$ | Numbers and Booleans |
| | | \| | $c \in \{\mathsf{Red}, \mathsf{Blue}, \mathsf{Green}, \mathsf{Pink}, \mathsf{Black}\}$ | Colours |
| | | \| | $x \mid \backslash x.e \mid \mathsf{rec}\, f(x).x$ | Anonymous functions |
| | | \| | $[v_1, \ldots, v_n]$ | Lists |
| | | \| | $(v_1, v_2)$ | Pairs |
| | | \| | $\ell$ | Reference locations |
| | | \| | $()$ | Unit value |
| 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 |
| | | \| | $c \in \{\mathsf{Red}, \mathsf{Blue}, \mathsf{Green}, \mathsf{Pink}, \mathsf{Black}\}$ | Colours |
| | | \| | $x \mid \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$ | Binding |
| | | \| | $\ell \mid \mathsf{ref}\, e \mid !e \mid e_1 := e_2$ | References |
| | | \| | $e_1\, e_2$ | Function Application |
| | | \| | $[] : \tau \mid e_1 :: e_2 \mid \mathsf{case}\, e \{[\,] => e_1 \mid x :: y => e_2\}$ | Lists |
| | | \| | $(e_1, e_2) \mid \mathsf{fst}(e) \mid \mathsf{snd}(e)$ | Pairs |
| | | \| | $() \mid e_1; e_2$ | Unit, Sequencing |
| | | \| | $\mathsf{while}\, (e_1)\, \{e_2\} \mid \mathsf{do}\, \{e_1\}\, \mathsf{while}\, (e_2)$ | Looping |
| | | \| | $\mathsf{forward}\, e \mid \mathsf{backward}\, e \mid \mathsf{right}\, e \mid \mathsf{left}\, e$ | Turtle Movement |
| | | \| | $\mathsf{penUp} \mid \mathsf{penDown} \mid \mathsf{setCol}\, e \mid \mathsf{randCol}\, e$ | Turtle Pen Control |
| | | \| | $\mathsf{let\ fun}\, f(x : \tau) = e_1\, \mathsf{in}\, e_2 \mid \mathsf{let\ rec}\, f(x : \tau_1) : \tau_2 = e_1\, \mathsf{in}\, e_2$ | |
| | | \| | $\mathsf{let}\, (x, y) = e_1\, \mathsf{in}\, e_2$ | Syntactic Sugar |
| Types | $\tau$ | ::= | `int` \| `bool` \| `color` \| `list`$[\tau]$ \| `ref`$[\tau]$ \| `unit` | |
| | | \| | $\tau_1$ -> $\tau_2 \mid \tau_1 * \tau_2$ | |

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

draws a square fractal.

## 6.3 Typing Rules

The typing rules for arithmetic, booleans, 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 Turtle, 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 pair values, list values, function values, or reference location values in an expression when it is being typechecked.

The Turtle-specific rules in Figure 9 for sequencing, while, references, and colors are new, but should also be fairly straightforward to implement.

---

**Exercise 3.** *Implement the typechecking function:*

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

*You may reuse cases from the typechecker for Giraffe from Assignment 2 (where appropriate).*

*[4 marks]*

---

## 6.4 Substitution and Desugaring

Capture-avoiding substitution is needed for $\lambda_{\mathsf{LOGO}}$ 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

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

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

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

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

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

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \texttt{->} \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \backslash x : \tau_1 . e : \tau_1 \texttt{->} \tau_2} \qquad \frac{\Gamma, f : \tau_1 \texttt{->} \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathsf{rec}\ f(x : \tau_1) : \tau_2 . e : \tau_1 \texttt{->} \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \mathsf{fst}(e) : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \mathsf{snd}(e) : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 * \tau_2 \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{let}\ (x, y) = e_1\ \mathsf{in}\ e_2 : \tau}$$

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

$$\frac{\Gamma \vdash e : \texttt{list}[\tau_1] \qquad \Gamma \vdash e_1 : \tau_2 \qquad \Gamma, x : \tau_1, y : \texttt{list}[\tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{case}\ e \{[\,] \texttt{=>} e_1 \mid x :: y \texttt{=>} e_2\} : \tau_2}$$

Figure 8: Typing rules for arithmetic, booleans, variables, functions, pairs, and lists

$$\frac{}{\Gamma \vdash () : \texttt{unit}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1; e_2 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{unit}}{\Gamma \vdash \mathsf{while}\ (e_1)\ \{e_2\} : \texttt{unit}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{unit} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash \mathsf{do}\ \{e_1\}\ \mathsf{while}\ (e_2) : \texttt{unit}}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{ref}\, e : \texttt{ref}[\tau]} \qquad \frac{\Gamma \vdash e : \texttt{ref}[\tau]}{\Gamma \vdash !e : \tau} \qquad \frac{\Gamma \vdash e_1 : \texttt{ref}[\tau] \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \texttt{unit}}$$

$$\frac{}{\Gamma \vdash c : \texttt{color}}$$

$$\frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \mathsf{forward}\ e : \texttt{unit}} \qquad \frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \mathsf{backward}\ e : \texttt{unit}} \qquad \frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \mathsf{right}\ e : \texttt{unit}} \qquad \frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \mathsf{left}\ e : \texttt{unit}}$$

$$\frac{}{\Gamma \vdash \mathsf{penUp} : \texttt{unit}} \qquad \frac{}{\Gamma \vdash \mathsf{penDown} : \texttt{unit}} \qquad \frac{\Gamma \vdash e : \texttt{color}}{\Gamma \vdash \mathsf{setCol}\ e : \texttt{unit}} \qquad \frac{\Gamma \vdash e : \texttt{list}[\texttt{color}]}{\Gamma \vdash \mathsf{randCol}\ e : \texttt{unit}}$$

Figure 9: Typing rules for sequencing, while, references, and Turtle constructs

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, references, and the Turtle Graphics 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* $\lambda_{LOGO}$ *expressions. (You may copy over the sample solution from Assignment 2 to use as a starting point.)*

*[2 marks]*

---

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

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

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* $\lambda_{LOGO}$ *with their desugared forms. (As before, you may use the similar cases for Giraffe as a starting point.)*

*[2 marks]*

---

## 6.5 Evaluation

The semantics of $\lambda_{\mathsf{LOGO}}$ is defined using large-step evaluation rules, parameterized by a *store* $\sigma$, that is, a map from reference locations to values. The evaluation judgment is as follows:

$$\sigma, e \Downarrow \sigma', v$$

where $\sigma$ is the initial store mapping reference labels to their values, $e$ is the expression to be evaluated, $\sigma'$ is the store after evaluation, and $v$ is the value.

The store only maps the reference locations to their values; we handle ordinary variable binding using substitution, as in the interpreters given in class (and differently from the environment-based semantics used in Assignment 2). Therefore, function values are of the form $\backslash x.e$ or $\mathsf{rec}\ f(x).x$.

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 difference is that when multiple subexpressions are evaluated, we need to pass the updated state obtained from evaluating the first subexpression into the evaluation of the second, and so on.

### 6.5.1 Imperative Constructs

Turtle includes "imperative" constructs such as references, sequencing, and while-loops.

For references, the expression $\mathsf{ref}\ e$ creates a new reference to the value of $e$, represented as a *location* value $\ell$, and adds a mapping from $\ell$ to the value of $e$ to the store. Locations are `Integers` and you can use the `Genloc.genloc()` method to generate a fresh location. The expression $e_1 := e_2$ evaluates $e_1$ to its (location) value $l$, evaluates $e_2$ to its value $v_2$, and updates the store so that $\ell$ is mapped to $v_2$. Finally, dereferencing $!e$ evaluates $e$ to its (location) value $\ell$, and looks up the value of $\ell$ in the store.

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

$$\frac{v \text{ is a value}}{\sigma, v \Downarrow \sigma, v}$$

$$\frac{\sigma, e_1 \Downarrow \sigma', v_1 \qquad \sigma', e_2 \Downarrow \sigma'', v_2}{\sigma, e_1 + e_2 \Downarrow \sigma'', v_1 +_{\mathbb{N}} v_2} \qquad \frac{\sigma, e_1 \Downarrow \sigma', v_1 \qquad \sigma', e_2 \Downarrow \sigma'', v_2}{\sigma, e_1 - e_2 \Downarrow \sigma'', v_1 -_{\mathbb{N}} v_2}$$

$$\frac{\sigma, e_1 \Downarrow \sigma', v_1 \qquad \sigma', e_2 \Downarrow \sigma'', v_2}{\sigma, e_1 * e_2 \Downarrow \sigma'', v_1 *_{\mathbb{N}} v_2} \qquad \frac{\sigma, e_1 \Downarrow \sigma', v_1 \qquad \sigma', e_2 \Downarrow \sigma'', v_2}{\sigma, e_1 / e_2 \Downarrow \sigma'', v_1 /_{\mathbb{N}} v_2}$$

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

$$\frac{\sigma, e_1 \Downarrow \sigma', v \qquad \sigma', e_2 \Downarrow \sigma'', v}{\sigma, e_1 == e_2 \Downarrow \sigma'', \mathsf{true}} \qquad \frac{\sigma, e_1 \Downarrow \sigma', v_1 \qquad \sigma', e_2 \Downarrow \sigma'', v_2 \qquad v_1 \neq v_2}{\sigma, e_1 == e_2 \Downarrow \sigma'', \mathsf{false}}$$

$$\frac{\sigma, e_1 \Downarrow \sigma', v_1 \qquad \sigma', e_2 \Downarrow \sigma'', v_2}{\sigma, e_1 < e_2 \Downarrow \sigma'', v_1 <_{\mathbb{N}} v_2} \qquad \frac{\sigma, e_1 \Downarrow \sigma', v_1 \qquad \sigma', e_2 \Downarrow \sigma'', v_2}{\sigma, e_1 > e_2 \Downarrow \sigma'', v_1 >_{\mathbb{N}} v_2}$$

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

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

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

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

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

$$\frac{\sigma, e \Downarrow \sigma', (v_1, v_2)}{\sigma, \mathsf{fst}(e) \Downarrow \sigma', v_1} \qquad \frac{\sigma, e \Downarrow \sigma', (v_1, v_2)}{\sigma, \mathsf{snd}(e) \Downarrow \sigma', v_1}$$

$$\frac{\sigma, e \Downarrow \sigma', v \qquad \ell \neq \mathit{locs}(\sigma')}{\sigma, \mathsf{ref}\, e \Downarrow \sigma'[\ell \mapsto v], \ell} \qquad \frac{\sigma, e_1 \Downarrow \sigma', \ell \qquad \sigma', e_2 \Downarrow \sigma'', v}{\sigma, e_1 := e_2 \Downarrow \sigma''[\ell \mapsto v], ()}$$

$$\frac{\sigma, e \Downarrow \sigma', \ell}{\sigma, !e \Downarrow \sigma', \sigma'(\ell)}$$

$$\frac{\sigma, e_1 \Downarrow \sigma', () \qquad \sigma', e_2 \Downarrow \sigma'', v}{\sigma, e_1; e_2 \Downarrow \sigma'', v} \qquad \frac{\sigma, e_1 \Downarrow \sigma', \mathsf{true} \qquad \sigma', e_2 \Downarrow \sigma'', () \qquad \sigma'', \mathsf{while}\ (e_1)\ \{e_2\} \Downarrow \sigma'''()}{\sigma, \mathsf{while}\ (e_1)\ \{e_2\} \Downarrow \sigma''', ()}$$

$$\frac{\sigma, e_1 \Downarrow \sigma', \mathsf{false}}{\sigma, \mathsf{while}\ (e_1)\ \{e_2\} \Downarrow \sigma', ()}$$

Figure 10: Evaluation rules

The other constructs support sequential composition and looping. The expression $e_1; e_2$ evaluates $e_1$, ignores its result, and then executes $e_2$. The while-loop $\mathsf{while}\ (e_1)\ \{e_2\}$ evaluates $e_1$ and if the result is true, evaluates $e_2$ and then loops, otherwise finishes.

### 6.5.2 Turtle-Specific Constructs

The semantics for the turtle movement and turtle pen control primitives are best described informally, as they are naturally side-effecting. The rules (not considering the side effects you will need to implement) will be of the following form:

$$\frac{\sigma, e \Downarrow \sigma', v}{\sigma, \mathsf{forward}\ e \Downarrow \sigma', ()} \qquad \frac{}{\sigma, \mathsf{penDown} \Downarrow \sigma, ()}$$

(and similarly for the other constructs),

They should firstly evaluate their arguments (if applicable), and then use appropriate `TurtleEDSL` operations to build up a `TurtleGraphics` value. As discussed below, we will maintain the `TurtleGraphics` value representing the behavior of the turtle as part of the program state in the evaluator; this is not made explicit in the above rules but should be handled similarly to the store $\sigma$.

We do not define evaluation rules for the let-operations or do-while operation, since they are syntactic sugar, which should be removed by `desugar` before evaluation.

## 6.6 Implementing the Evaluator

The `Eval` class provides a function `run` to evaluate whole $\lambda_{\mathsf{LOGO}}$ programs. The `run` method is located in a class `Eval` that takes an instance of the `TurtleDSL` trait as a parameter, and import it, so that we can run $\lambda_{\mathsf{LOGO}}$ using different instances of the `TurtleDSL` trait, such as `Testing` for testing or `TurtleDSLImpl` to actually create the graphics. The `run` function is given an expression, width, height and output filename, and does the bureaucratic work of setting up the graphics canvas. It calls a (private) helper function *eval* that takes a state and expression and yields an updated state and value. The state is of type `TurtleState`, which is defined as follows:

```
private final case class TurtleState(
  store: Store[Value],
  graphics: TurtleGraphics
)
```

Here, `Store[Value]` is the type of value stores, that is, mappings from `Locations` to `Values`. The second component is a `TurtleGraphics` value built up using the operations in the provided `TurtleDSL` instance passed in as an argument to `Eval`. The `eval` method should follow the inference rules given above, maintaining the reference state in `state` and adding the graphics operations evaluated by the program to `graphics`. The `TurtleGraphics` value in the final state will be rendered and saved to disk.

---

**Exercise 6.** *Implement the evaluator* `eval`*, whose type signature is as follows:*

```
def eval(state: TurtleState, expr: Expr): (TurtleState, Value)
```

*[8 marks]*

---

## Change Log

- V1.1 (November 9): Changed “abstract type” to “abstract class”
- V1.2 (November 14): Fixed typos in case, reference creation, and assignment rules.
- v1.3 (November 21): Added missing rule for typing ‘list construction’ or ‘cons’ ($e_1 :: e_2$), and fixed typo in the ‘nil’ rule.
- v1.4 (November 22): Added missing rule for evaluating division.
- v1.5 (November 23): Fixed a re-introduced typo in reference creation rule, and added evaluation rules for fst and snd.