PL Research: Language Specification

John Loeber
May 2, 2015

To avoid leaving the project nameless any further, I propose that the program (i.e. the software being developed) will be called **Embi** (as a working name). Let's recap the features of Embi:

1. Graphical interface ("GUI") for manipulating visual elements on the canvas, like in Paint, Photoshop, etc.
2. Data panel that stores the **data**: all information on all visual elements on the canvas. May be manipulated by the user.
3. Code panel that allows the user to modify the data using a domain-specific language, **Embil**.
4. History panel (tabbed with code) that records, in Embil, all changes made to the program state.

The purpose of this document is to lay out a language specification for a toy version of Embil (i.e. one that I can implement within the next few weeks). I'll need some guidance on the proper PL conventions regarding reduction rules, notation for typing, etc.

High-Level Stuff

In this draft, the code-panel is basically a REPL for Embil. Appropriately, Embil is simple. It currently does not support user-defined functions or types (though it should in a more mature version). As the language has a strong imperative/declarative touch to it and is meant to support experimentation, all variables are mutable. Types are not declared, but inferred.

Visual elements and functions generally take arguments. The most common arguments are the (x,y) coordinates of an object. These (x,y) coordinates are **always** with respect to the top-left corner of the object. Similarly, the top-left corner of a canvas is (0,0), and the bottom-right (x,y).

Here's what's in toy-Embil. I also want to make a list of the things I want in full-fledged Embil (user-defined functions, typeclasses, Haskell- and Python-style list comprehensions, etc.), but I'll hold out on that until the toy-version is ready.

Note that for properties of objects, I like an explicit syntax à la `object(property:value)`. If the user does not enter a value for a property, that property defaults to a default value. Defaults are to be decided, as are syntactical conventions. This is not yet a thorough documentation at all, just some notes on what things should be like.

Datatypes

1. Integers
2. Floats
3. Strings (currently only to describe colors)
4. Lists: `[] [1,2,3]`
5. Visual Elements
   (a) Rectangles: `rect(w, h, x, y, color, opacity, rotation, xstretch, ystretch)`
   (b) Circles: `circle(r, x, y, color, opacity, rotation, xstretch, ystretch)`
   (Note: the rotation option is relevant if the circle was stretched in any direction.)
   (c) Spaceholders: `space(w,h,x,y)`
   (Effectively a transparent rectangle.)

   Note that modifying the `xstretch` and `ystretch` properties will change the width and height of an object, but **not** the x and y coordinates. This means that stretching an object will also translate it – the user will have to adjust for this. (It would be easy to nest modifying functions to achieve this goal.)
6. Groups: special lists in which each element is unique (like in a set in other languages). Moreover, a group is either (a) a list of visual elements, (b) a list of groups, or (c) the empty list.

Operations on Integers and Floats

$+ \ - \ \cdot \ / \ \% \ a^b \ \log_b a$ sin cos. The usual pre-calculus operations.

Operations on Strings

None

Operations on Lists

1. Access the ith element: `List[i]`
2. Append x to a list: `List.append(x)`
3. Delete the ith element from list: `List.delete(index:i)`
4. Delete the element of value x from list: `List.delete(element:x)`
5. Test if x is in List: `List.member(x)`
6. Length of a list: `List.length()`
7. Map a function over a list: `List.map(function)`
8. Generate a list of ints/floats: `range(a,b,c)` Where a is the start, b is the end, and c is the step size. E.g. `range(0,5,1) = [0,1,2,3,4]` and `range(0,1,0.2) = [0,0.2,0.4,0.6,0.8]`. Default step size is 1.
9. Iteration: the user may only iterate over lists. `for item in list: do something`. Thus, traditional for-looping over integers as with `for i=0;i<n;i++)` is still possible, but would use syntax `for i in range(0,n)`. (This is python-inspired.)

Operations on Visual Elements

1. Generate n distinct copies of a VE: `VE.generate(n)`
2. Modify a property of a VE: `VE.modify(property:newvalue)`
   Where if you want to e.g. increment a property by 10, it is correct to write `VE.modify(property:VE.property+10)`. Several properties may be modified at once. (E.g. `VE.modify(p1:5,p2:60,...)`. Note that the user would be expected to just manipulate the data rather than write this command. This command would be the output to the history when a user manipulates the data and propagates the change.
3. Resize a VE: `VE.resize(proportions:bool, wpercent : int or float, hpercent: int or float, hpx: int or float, hpx: int or float)`
   Where `proportions` is a bool with which the user may choose whether to keep or discard the current width/height ratio of the VE. The user may specify the new width or height in either percent (relative) or pixels (absolute). If both width and height are specified, then proportions defaults to false.

4. Addition: occasionally it may not be convenient to combine VEs as groups. So we can simply combine visual elements by `VE3 = VE1 + VE2`.

Operations on Groups
(Note that since Groups are Lists, they inherit methods like length, member, etc.)

1. Intersection: `intersect(group1,group2,...groupn)`
2. Union: `union(group1,group2,..., groupn)`
3. Complement: `group1.complement`
   (This is every element in the data that is not in group1.)
4. Spacing along the x-axis: `group.xspacing(int or float: value, sequence: list)`
   Where the sequence would be a list of $n-1$ floats or ints for a group of size n
5. Spacing along the y-axis: `group.yspacing(int or float: value, sequence: list)`

Other Operations

1. `initialize canvas(w,h)`
   Initializes the canvas with dimensions (width,height).

Temporary and Persistent Variables

We've discussed temporary and persistent variables in Embil previously. The idea here is that we want to minimize co-dependencies between objects in the code. All variable declarations are persistent. Anything else is temporary, i.e. discarded after propagation. I need to figure out some way to have the `generate` statement stand on its own and discardable, while at the same time creating a group from the generated elements – I generally need to figure out a way to allow the user to easily create groups. Perhaps some sort of list syntax would do it? E.g. `group([element1,element2,...,elementn)`