Unifying the Best Features of Graphics Languages

Jake Barnwell & Miles Steele

6.905/6.945: Large-Scale Symbolic Systems
2015 May 4

1. Problem Domain

There are many possible languages for describing images. Different graphical description languages can describe the same image but in different ways and through different lenses. Some utilize recursion to easily draw self-similar shapes or patterns while others use a stack of transforms to a similar effect, and still others use a step-by-step procedural methodology. We explored these different approaches through three languages designed for graphical programming and attempted to unify the approaches. During this exploration, we restrict ourselves to simple geometric line drawings..

2. Overview

The original intent of this project was to be able to explore the similarities of different graphical languages, and attempt to unify them in a system somehow. In addition, we wanted our system to easily allow a user to draw what they wanted, and also easily allow a programmer to easily update the system, potentially adding new features to it. In particular, we took two graphics languages, Logo[1] and Context Free,[2] and translated them (with some minor modifications) to be useable in MIT/GNU Scheme. Each language has its own interpreter which translates the

[1] See http://en.wikipedia.org/wiki/Logo_%28programming_language%29
[2] See http://www.contextfreeart.org/mediawiki/index.php/Context_Free_Art:About

requested graphics into a *Uniform Representation (UR)*, so-called because the representation of the graphics data in the UR was independent of what language it came from.

Once there, we further implemented two separate graphics backend interpreters, X Graphics and SVG. Each graphics interpreter can read from a UR data structure and draw the appropriate shapes to the screen.[3]

In doing this, we came across several features of Logo and Context Free that we liked (or disliked) and decided to design our own graphical language--one slightly more suited to Scheme--that exhibited the "good" (likeable) properties of Logo and Context Free, while avoiding the bad ones. This language, which we will refer to as the hybrid language, also utilizes the UR. All graphics written in the hybrid language can also be translated to a UR and similarly displayed via X Graphics or SVG (Figure 1).

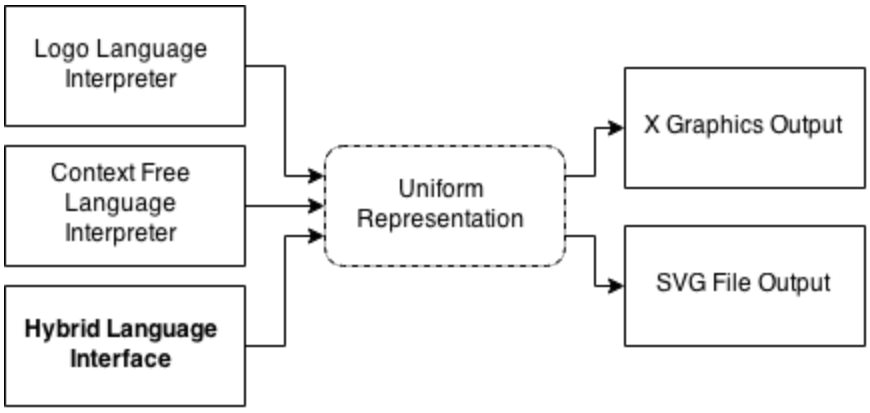


Figure 1. System Dataflow. We can take any of three input languages and transform them into the Uniform Representation. From there, two different backends may be used to interpret and draw the shapes given in the UR.

3. Uniform Representation

As a way of cleanly modularizing our system and separating the front-end components from the back-end components, we have developed an intermediary representation (Uniform Representation or UR) that stores all of the data necessary to invoke drawings on a canvas.

A UR dataset is a list of commands, and each command is of the form
`(<feature> args…)`
where `<feature>` is one of `point`, `line`, or `color`.[4]

In particular, the following are valid commands:

[3] Technically, the SVG backend doesn't support points. This is one thing we are still working on.
[4] The SVG backend does not support the color command; it simply ignores any color command it sees.

- `(point <x> <y>)` represents a point at location (x, y)
- `(line <x1> <y1> <x2> <y2>)` represents a line segment from point $(x1, y1)$ to $(x2, y2)$
- `(color "<color-name>")` implores that any shapes drawn after this command be of color `<color-name>`, until a new color command is invoked

4. Front End Languages

We integrated three swappable graphical languages into the front end of our system: this gives a user the choice between three different input styles, depending on what they are trying to accomplish. The creation of the interpreters for the first two languages, Logo and Context Free, informed our design and creation of the third custom hybrid language.

4.1 Logo Interpreter

The Logo language is a language for writing programs that draw graphics. It was designed in the 1960s by Seymour Papert and others [2, 3]. We implemented an interpreter for a miniature version of Logo with fewer features and a Scheme-like syntax that outputs to our Uniform Representation.

The basic idea of Logo is that there is a single turtle with a pen on a canvas. Commands can be issued to the turtle and it will move around and be implored to draw things. We can define a procedure with the keyword `to`, thus telling a turtle how to do some re-usable task.

The turtle knows several primitive commands:

- Move forward, written as `(forward 10)` or `(fd 25)`
- Turn right by some number of degrees: `(rotate 90)` or `(rt 10)`
- Pick up or lower its pen to start/stop drawing: `(pen-up)` or `(pen-down)`
- Set its pen's color: `(color "blue")`

We can teach the turtle new tricks by defining procedures. Here we tell the turtle how to draw a square of a given size.

```
(to (square size)
  (repeat 4
    (fd size)
    (rt 90)))
```

Telling the turtle how to do something causes no change; the change is only invoked by executing the procedure.

```
(square 100)
```

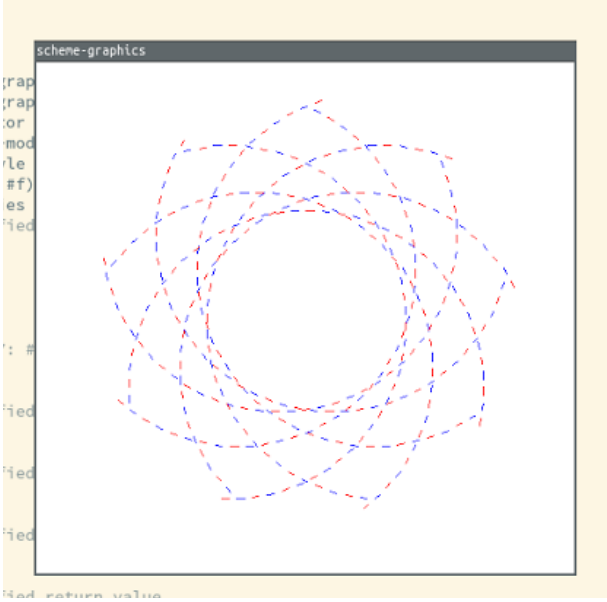
There are three ways to use the Logo interpreter to create images. All three produce Uniform Representation objects which can be sent to the X or SVG backends.
To evaluate a quoted expression directly from scheme, we use `logo:eval`. This is the lowest level interface to the interpreter. A usage example is as follows:

```
(define canvas (logo:canvas:new))
(define env (logo:make-env))
(logo:eval '(fd 20) env canvas)
(logo:canvas:turtle canvas) ; => (turtle (100 0) 0 #t)
(logo:canvas:ur canvas) ; => ((line 0 0 100 0))
```

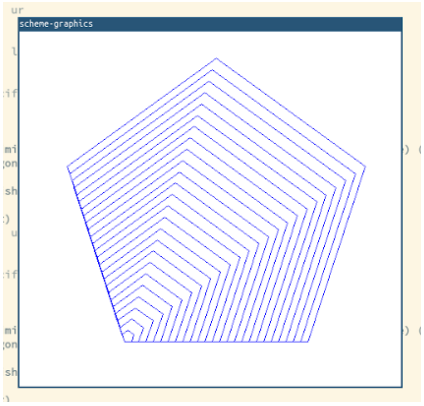
To evaluate a file containing a Logo program, we use `(logo:eval-file "program.logo")`.

The third way to run the Logo interpreter is through the repl. Call `(logo:repl)` and we enter a read-evaluate-print-loop for Logo until an error occurs or `(commit)` is entered, at which point the `(logo:repl)` call will return the logo canvas:

```
;;; Draw from REPL input.
(define ur (logo:canvas:ur (logo:repl)))
;;; Into REPL
(to (dotted-line length)
  (repeat (/ length 10)
    (pen-up)
    (fd 5)
    (pen-down)
    (fd 5)))
(to (curve)
  (repeat 10
    (repeat 2
      (color "RED")
      (dotted-line 10)
      (color "BLUE")
      (dotted-line 5)
      (rt 10))))
(repeat 10
  (curve)
  (pen-up)
  (rt 180)
  (fd 100)
  (pen-down)
  (rt 180))
;;; (commit) terminates the repl and returns the canvas
(commit)
(draw ur) ;; Draw using the X backend
```



Here is another example which uses automatic recursion base-case detection, an idea borrowed from the Context Free language (see Section 4.2). This piece of code uses `ngon` which can be found in `logo-library.logo` which is automatically loaded into the Logo repl. It also uses the `limit` form which stops recursive functions from running forever. When the limit is reached, the function is terminated using a continuation at the invocation level so that sister calls will still occur.




```
(to (ngon sides size)
  (repeat sides
    (fd size)
    (rt (/ 360 sides))))
(to (diminishing-ngons sides size)
  (limit size 0.01)
  (ngon sides size)
  (diminishing-ngons sides (- size 10)))
(diminishing-ngons 5 250)
```

For drawing pictures, this mini-Logo language has a few good features. It makes it easy to encapsulate reusable pieces of images. For example, in the first example, the same `dotted-line` and `curve` shapes are used many times with different parameters. Reusable procedures can take parameters to be used for different tasks. A mixed blessing of Logo is that every movement is permanent. So if the turtle is moved forward and turn many times in such a way that it does not naturally end up back where it started (as is the case for polygons), it is difficult to get the turtle's position to behave predictably. This means that drawing scenes where objects are in specific places is inconvenient, because each the author of each procedure would have to be responsible for moving the turtle back to where it started so that the next procedures could behave predictably. On the other hand, drawing this way can be very intuitive, as one doesn't have to think about transformation or matrices, and instead just get the turtle meandering along.

4.2 Context Free Interpreter

We use the term Context Free to refer to the language that is technically known as *Context Free Design Grammar (CFDG)*, first developed by Chris Coyne[5] [1]. CFDG is, per its name, a context-free language, which means that the rules for expanding and evaluating symbols are constant. That is, evaluation of a command or symbol occurs deterministically and independently of any other commands or symbols in the program.

The Context Free language has very few primitives, but is extremely powerful. Indeed, excepting control structures and color manipulation, there are only three primitive shapes and a few primitive commands.[6]

[5] Context Free actually refers to the application in which the user writes the code and displays the resultant images.

[6] For a full overview, see http://www.contextfreeart.org/mediawiki/index.php/CFDG_HOWTO

Every program starts with the user declaring a "starting shape." The user then declares rules for the starting shape; if multiple rules for a single shape are declared, the user can assign weights to the rules such that one rule is randomly selected from the set. Each rule can include drawing one of three primitives (circle, triangle, or square), drawing another user-defined shape, or even recursively calling the same shape. The user also specifies what transformation parameters should be applied to each entity being drawn, such as scaling, rotation, or translation.[7] And, like most other programming languages, users can create functions and variables.

At execution, the interpreter attempts to draw the starting shape **S** by looking for any rules of **S**. If there are multiple rules supplied, one is randomly chosen depending on what weights were assigned to the rule. Each command in the chosen rule is interpreted in order, and any shapes found are drawn; this might in turn call more draws recursively on the same or other shapes.

Our implementation works with a slightly reduced set of features. However, most of the important ones were kept.[8]

The general form of a program is as follows:

```
(startshape S)
(let const1 expr...)
(let const2 expr...)
...
(shape S ( rule… ) ( rule… ) … )
...
(shape T … )
...
```

A more concrete example will be provided later.

In particular, the following commands are supported in our Scheme variant of Context Free:

- `(startshape S)` declares **S** to be the startshape. There must be exactly one `startshape` command in the program.
- `(startshape S (`*params…*`))` declares **S** to be the startshape with certain transformation parameters (rotate, scale, translate, or flip). There must be exactly one startshape command in the program.
- `(shape S` *rule*`)` assigns to **S** the rule *rule*, which is always interpreted on the fly whenever a command to draw **S** is invoked.
- `(shape S (rule <?num>` *rule-1*`) … (rule <?num>` *rule-n*`))` is similar except for one of *rule-i* is randomly chosen according to weights specified by `<?num>`. If no weight is

[7] There are also a variety of options for changing the colors and transparency.
[8] We were not able to implement functions or colors, for example.

supplied, it is assumed to be `1`. The weights are normalized over their sum[9] and a corresponding rule is chosen accordingly. Note that a rule is randomly chosen at *each* invocation of **S**, and not pre-computed. Having more than one `shape` command for a shape **S** produces undefined results.

- `(<shapename>)` or `(<shapename> ())` makes an invocation to draw the shape named `<shapename>` with no additional transformation parameters. If the requested shape is a primitive (`triangle`, `circle`, or `square`), the drawing is immediately applied. If the requested shape is user-defined, the interpreter searches for applicable rules for the shape and applies them.
- `(<shapename> (params...))` similar to above but requests to draw the shape with the given transformation parameters. Note that transformations compound every time another shape is called with a transformation.
- `(let c expr)` assigns the value of *`expr`* to the constant `c`. The constant `c` is constant and hence may not be re-assigned at a later date.

Possible transformation parameters are listed below:

`x <n>`	Translate by `<n>` in the x
`y <n>`	Translate by `<n>` in the y
`{t, translate, trans} <m> <n>`	Translate by `<m>` and `<n>` in x and y, respectively
`{s, scale, size} <m> <n>`	Scale by `<m>` and `<n>` in x and y, respectively
`{dr, drotate, drot} <n>`	Rotate counterclockwise by `<n>` degrees
`{rr, rrotate, rrot} <n>`	Rotate counterclockwise by `<n>` radians
`{flipx, fx}`	Flip across the x-axis
`{flipy, fy}`	Flip across the y-axis
`{dflip, df} <n>`	Flip across the line that goes through the origin and is `<n>` degrees above the x-axis
`{rflip, rf} <n>`	Flip across the line that goes through the origin and is `<n>` radians above the x-axis

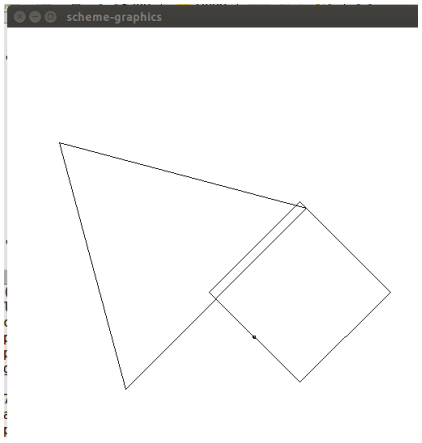
A transformation params list is just a concatenation of any transformations that are desired, in order. For example, the transformation list

```
(s 0.5 1.5 x 1 dflip 45 translate 1 2)
```

[9] For example, if the weights were (1, 0.5, 1.5), then the evaluator would pick the first rule with probability ⅛, the second with probability ⅛, and the third with probability ½.

requests that the following transformations be done to a shape, in order: scale it by 0.5 in the x-direction and 1.5 in the y-direction, then translate it 1 unit to the right, flip it across the $y = x$ line, then translate it (again) by 1 in the x-direction and 2 in the y-direction.

There are a few things of note. The first is that transformations are applied cumulatively down the call chain. For example, consider this code snippet:




```
(startshape foo (dr 45))
(shape foo (
             (triangle (y 0.5))
             (bar (s 0.5 0.5))
           ))
(shape bar (
             (square (x 0.5))
            ))
```

The transformation "rotate 45 degrees" given in the `startshape` command is applied first, and applies to *everything drawn under the umbrella of* `foo`. Hence, a triangle is drawn 0.5 units above the origin *in the rotated space*, i.e. 0.5 units to the upper-left of the original origin. Similarly, invoking `bar` with the scale parameters sets the upcoming frame of reference to be not just rotated 45 degrees, but also shrunken by a factor of two. Hence, the square is drawn at half size, and is located 0.5 units to the right in the new frame of reference (i.e. 0.5 units to the upper-right of the origin). We can easily see this in the image above (the black dot is the origin).

Second, all transformations are stored as matrices, and as we progress down the call chain, we multiply the transformation matrices. For any given primitive, the resultant matrix correctly represents the sequence of transformations. The astute reader will have noted that translations cannot typically be represented as matrices: translations are affine transformations with no fixed points, meaning a 2D translation cannot be represented with a 2x2 matrix. To avoid this issue, we do all transformations with 3x3 matrices and use homogeneous coordinates for the points (vectors).[10]

Lastly, we note that recursion is not only possible, but also highly encouraged. If, for example, a shape is declared as

```
(shape rec (
             (square ())
              (rec (s 0.9 0.9))
             ))
```

then attempting to draw foo will cause it to recurse on itself. What makes Context Free unique is that there are no explicit base-cases; assuming that the size of some shape converges to zero in

[10] See http://mathworld.wolfram.com/HomogeneousCoordinates.html

some dimension, the interpreter will know this and will halt drawing when the shape becomes too small. We do this by checking that the transformation matrix is still "large enough" to make the image be seen if it were drawn; if not, we halt.[11]

In order to execute and draw a program in this style, we have to call the function ctxf as follows:
`(ctxf '( commands ... ) )`
We will see an example of this below.

We now present a short example of recursive code:

```
(ctxf '(
            (startshape branch)
            (shape branch
                   (rule 3 (
                            (branch (dr 5 s 0.95 0.95))
                            (branch (dr -25 s 0.95 0.95))))
                   (rule 99 (
                       (triangle (s 0.4 0.4))
                       (branch (dr 1 y 0.13 s 0.97 0.97))))
             )
       ))
```

The shape branch always picks between two rules: the first with roughly 0.03 probability, and the second with approximately 0.97 probability. The second rule continually draws triangles forward while slowly rotating left, while the first rule splits into two branches that recursively draw themselves. See Figure 2 for some different outcomes of this program. We note that the outcomes are very different. If the first rule can be picked early, there's a lot higher chance for more branch offshoots; however, if it takes awhile for the first rule to be picked, the tree will still die off very quickly.

[11] In particular, we take three known non-collinear points and transform each of them with the transformation matrix. If the distance between any two of them is below a certain threshold, then it's likely that the transformation matrix has dwindled and will continue to zero in that dimension, and we can halt drawing.

Figure 2. Randomized Trees via Context Free Interpreter. Note how different the trees look from each other. This is due to the randomness in the branching algorithm.

Over the course of this project, we have grown to enjoy several of the features that CFDG and our variant offer. One of the most important of these is the self-halting recursion (a feature so important that we ported it to our Logo implementation!). Allowing the interpreter to handle recursion base-cases makes the user's life much easier when writing code. Furthermore, the code is easier to read and debug.

We also believe that the compounding transformation model is important, as it is something easily intuited by users. This also allows easy recursion since each child inherits its parents transformations. Lastly, set of transformations acts as a stack; if there is a branching recursion, transformations are effectively popped from the stack as the recursion stack unravels, meaning that different branches of recursion behave, as expected, independently of each other but dependent on their ancestors.

One of the downsides we found to Context Free is that, while its ability to generate recursive or abstract images is impressive, it is much more difficult to generate an arbitrary drawing or shape. For example, if one wanted to draw a half-moon or a rectangle with a corner cut off, it would be relatively easy to do so in Logo, but not in Context Free, since CFDG only allows the creation of three primitive shapes, relying on image transformations to create different outputs.

4.3 Hybrid Language

The third language we present is a hybrid language in which we attempted to combine the full-featured power of Scheme with our favorite parts from both Logo and Context Free. Instead of writing an interpreter, we implemented as much as we could native to Scheme, which means that all of the familiar and powerful features of Scheme are available in the hybrid language, as it is really just an interface for building a Uniform Representation. From Logo we kept procedures

as well as the option of having permanently applied transformations (analogous to movement in Logo). From Context Free we took the idea of an active transformation stack which can be unwound. From Context Free we also take automatic recursion base-case detection. We can demonstrate the power of this hybrid language by implementing exciting operations such as `mirror` even on procedures which are made with permanent transformations in mind.

To use the hybrid language, just load `hybrid.scm`. The primitives are as such:

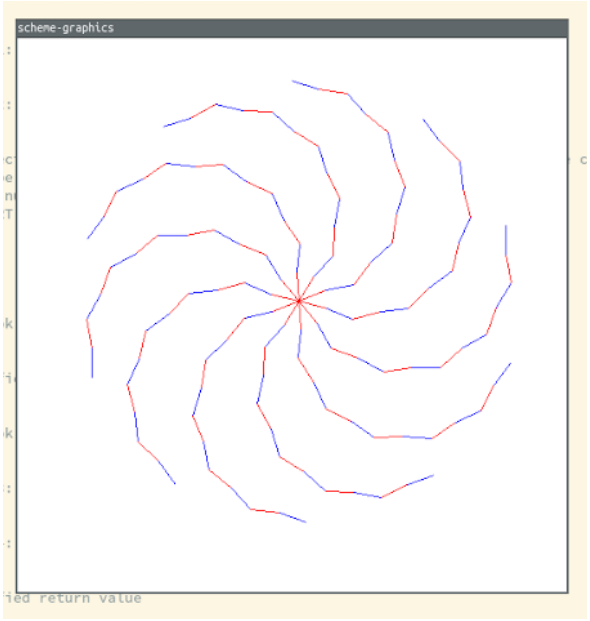
- `hybrid-reset!` clears the current canvas
- `*ur*` holds the active Uniform Representation, which can be drawn for with the X backend by running `(draw *ur*)`
- `(line! x1 y1 x2 y2)` draws a line
- `(color! colorstr)` which sets the active color
- `translate rotate scale forward` take an optional thunk to do within the transformation. If a thunk is provided, the transformation is unwound after the thunk occurs with the transformation. This is like Context Free. If no thunk is provided, the transformation is applied "permanently" to the active transformation.
- `save-excursion` is used in part of the implementation of the transformations, and is a powerful component of the system. The name and idea behind this is taken from emacs LISP. `save-excursion` runs a thunk within an isolated transformation stack so that any transformations that occur within the save-excursion are unwound when it finishes.

Here are some examples of how to use the hybrid language:

```
(hybrid-reset!)
(define (square2 size)
  ;; repeat takes a thunk to do n times in a row.
  (repeat 4 (lambda _
    (line! 0 0 0 size)
    (forward size)
    (rotate 90))))
(color! "black")
(square 100)
(draw *ur*) ;; shows a square
```

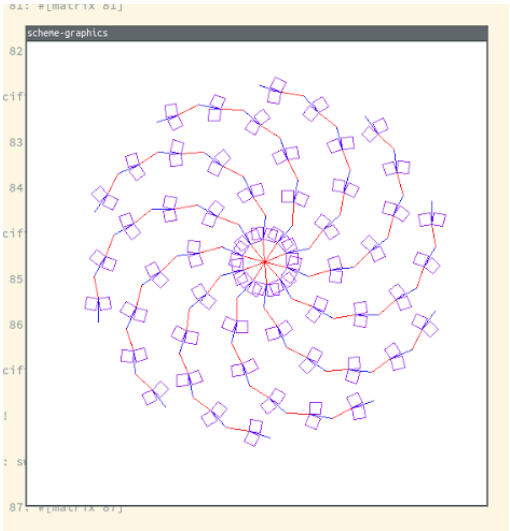
Here we start to see the advantage of mixing both semi-permanent transformations from Logo and unwindable transformations from Context Free. `repeat` allows each round of the repetition to affect the following rounds: but outside of the `repeat`, the transformations are unwound.

```
(hybrid-reset!)
(repeat 10 (lambda _
  (repeat 6 (lambda _
    ;; inside this inner repeat, the
    ;; transformation is advanced,
    ;; and that affects the
    ;; next line segment
    (rotate 40)
    (color! "red")
    (line! 0 0 0 100)
    (forward 100)
    (rotate -10)
    (color! "blue")
    (line! 0 0 0 100)
    (forward 100)))
  ;; But the out here in the outer
  ;; repeat, all the
  ;; transformation of the inner
  ;; repeat is unwound.
  ;; So the inner repeat doesn't have to be responsible
  ;; for returning the would-be turtle to the center again.
  (rotate (/ 360 10))))
(draw *ur*)
```



We can easily decorate shapes by passing shape parameters. Here is an example in which the previous is encapsulated in a function and is passed a shape parameter which it runs at each bend.

```
(hybrid-reset!)
(swirl (lambda _
  (color! "purple")
  (mirror 80 (lambda _
    (square 50)))))
(draw *ur*)
```



The `mirror` command above is easy to implement in the hybrid language:

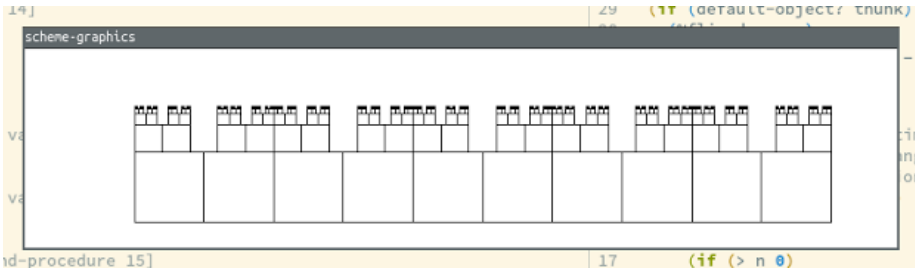
```
(define (mirror degrees thunk)
  (save-excursion thunk)
  (flip degrees (lambda _
    (save-excursion thunk))))
```

Where `flip` is wrapper like `translate` and `rotate` over one of the primitive matrix transformations. It can behave either like Logo with a persistent application to the active transformation (when no thunk is provided) or like Context Free with an unwindable application (when a thunk is provided).

```
(define (flip degrees #!optional thunk)
  (if (default-object? thunk)
    (%flip degrees)
    (save-excursion (lambda _
      (%flip degrees)
      (thunk)))))
```

Here is a demonstration of automatic base-case detection, a key feature from Context Free, which is also available behind the `guard` helper in our hybrid language.

```
(define (castle-walls)
  (mirror 90 (lambda _
    (square)
    (translate -0.6 1)
    (scale 0.4)
    (guard
castle-walls))))
(hybrid-reset!)
(mirror 90 (lambda _
  (repeat 3 (lambda _
    (castle-walls)
    (translate 2 0)))))
(draw *ur*)
```



Note that `castle-walls` is recursive with no base case, but it does not run forever. The use of `guard` to call the recursive function causes it to be called cautiously. This is the same idea as Context Free's implicit termination and the `limit` extension to Logo. `guard` cause a continuation to be installed and when a graphics primitive like `line!` notices that it is drawing something deemed too small to see with the active transformation,[12] it short-circuits the function to avoid an infinite recursion. Notice though, that this is better than just throwing an error or aborting, because other sister leaves of the tree being drawn are not affected. Hence, much like in our Context Free implementation, this method works for “multi-pronged” recursion, as is illustrated in the castle walls image above.

In comparing the hybrid languages to the other languages we explored, the hybrid language has several advantages for easily expressing graphics. It can easily be used in just the same manner as Logo or as Context Free by just ignoring its extra features. Then, on top of programs written in that style, combinations can be applied in the form of operations like `mirror` which both use active transformations and take arguments which can be shapes themselves.

The hybrid language has some downsides, however, mostly in the form of clutter. There are many ugly `lambda` keywords and extra parens lying around which don't add to the programs’ utility. These could be avoided by implementing the likes of rotate as `macros` instead of as normal scheme procedures, but that might damage the composability. The other major wart of the hybrid

[12] via a small, arbitrary constant as a threshold

language is the `guard` must be used explicitly when calling potentially-recursive functions. It would be better if this could be implicit, as it is in Context Free and our Context Free implementation.

5. Back End Graphics

We have two backend graphics systems for dealing with shapes and displaying them to the screen: the X Graphics system, and SVG.

5.1 X Graphics

Our original method for displaying drawings uses Scheme's built-in graphics, which use the X Graphics system. Implementing this was not that difficult, especially since our UR only stores point and line information (and color, of course). We only need to loop through each of the elements in the UR and apply a draw method on it. In addition, we have implemented an auto-scaling feature that automatically resizes the canvas to display everything that has been requested to be drawn on the canvas.

One additional feature that we were able to implement with X was the ability to "slow-draw," allowing the user to watch as shapes are drawn real-time. We do this by breaking up every line into several thousand points and individually drawing the points with delays between each draw. [13] This feature is particularly interesting when paired with recursive programs: the user can watch as the recursive tree (or wall, or whatever) is built in real-time.

However, the slow-draw feature does need some work. If there are lots of entities that are being drawn but cannot be seen,[14] the system may spend a long time "drawing " them with no visible results. Furthermore, the speed is not dynamic; at the moment, we have hard-coded the delays between drawings and therefore longer URs may take inordinately large times to draw and are just not very generalizable to different-sized URs.

5.2 SVG

The SVG backend located in `svg.scm` consumes a UR object and produces an svg file. This backend is severely underdeveloped, but still has basic capabilities. It supports lines and automatically resizes the UR to fit (just like for the X backend) in a fixed-size svg file. Unfortunately, SVG does not support drawing points or colors, and seems to have a few bugs in it as well.

[13] We know this is very inefficient. However, we have not noticed any decrease in performance.
[14] This often happens with recursive programs, when system cannot very accurately determine when it is time to stop drawing entities because they are too small to see.

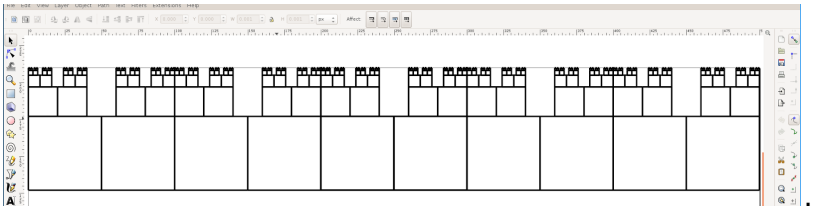


Figure 3. Using SVG to Draw. This is an example of the `castle-walls` drawing being displayed with SVG

6. Conclusion

In our efforts to design and implement our versions of both Context Free and Logo we have come to believe that the following are some of the more important features of a usable graphical description language:

1. An active transformation state which affects future operations.
2. Persistent transformations for easy chaining of commands.
3. An unwindable stack of transformations for isolating subroutines.
4. Automatic recursion base-case detection for drawing self-similar pictures.

We attempted to take these features and combine them into a cohesive hybrid language which exhibits the useful features of both languages, as well as some of the power of higher-level combinators. In doing so, we introduced some more language warts and confusion, but on the whole have produced a viable proof of concept that the ideas of forward-procedural and unwindable stack-based graphical paradigms can coexist peacefully in one language.

7. Future Work

Although we were immensely successful in emulating the behaviour of Logo and Context Free, as well as designing our own custom language, there are a couple of issues that persist. The first is that the hybrid language has a lot of unnecessary lambda-clutter in it, due to its functional nature. It would be ideal if we could remove a lot of this overhead, perhaps by using macros. In addition, there are several features that we know we are lacking, for example, drawing points or using color with the SVG backend.

Lastly, we want to explore these languages in the context of other types of drawing, not just geometric recursions or abstract art.

8. References

[**1**] "Context Free Art: About." *Context Free Art*. 20 Jan. 2008. Web. 4 May 2015. <http://contextfreeart.org/mediawiki/index.php/Context_Free_Art:About>.

[**2**] "Logo Language." *Logo Language*. Cunningham & Cunningham, 11 Sept. 2014. Web. 4 May 2015. <http://c2.com/cgi/wiki?LogoLanguage>.

[**3**] "What Is Logo?" *What Is Logo?* MIT, 1 Jan. 2011. Web. 4 May 2015. <http://el.media.mit.edu/logo-foundation/logo/>.