UNIVERSITY OF THE WEST INDIES
Department of Computing
COMP3652–Language Processors
Sem. I, 2015
Lecturer: Dr. Daniel Coore

Assignment 3: Due Thursday, Nov. 26, 2015

Introduction

In this assignment, you will build on the work you did in the previous assignment to develop a translator from *HPL+* to SVG.

A new package called `hpl.comp` has been provided for you, which should contain any new classes that you develop that are specific to the compiler. Two interfaces have been provided for you in that package already. They are: `HPLCompiler` and `CompilerResult`. These interfaces are quite simple, but are useful for allowing a top level class to perform generally useful tasks, such as accepting a file name as input, reading it in, parsing it, invoking the compiler on the resulting tree to produce an instance of `CompilerResult`, and then finally outputting the contents of the result to a file.

In all likelyhood, you will find it convenient to implement both the `HPLCompiler` and the `Visito<S, T>` interfaces with the same class. This is perfectly acceptable. The only reason that `HPLCompiler` is not a sub-interface of `Visitor<S, T>` is that keeping them separate allows a top level class such as `hpl.sys.HPLComp` to reference an instance of `HPLCompiler` without having to make any assumptions about the type of visitor context (`S`) that is required for it.

Problem 1 [50]
Implement a class called `HPLToSVG` that implements both the `HPLVisitor` and the `HPLCompiler` interfaces.

You may have to implement more than one class that implements the `CompilerResult` interface. For example, if may be a good idea to represent each type of SVG output (whether script fragment or SVG sub-element) with a separate class, all of which inherit from some parent class that implements the `CompilerResult` interface. This way, the `CompilerResult` output is a sort of tree, whose `gen` method will output the final contents of the output file. How you define this family of classes is left entirely up to you, because it will be different, depending on the approach that you choose to take for compilation. (See the section with the example later in this document, for an illustration of two of the approaches.)

Background on SVG

SVG is an XML standard for rendering vector graphics. It provides basic primitive graphical elements such as rectangles, ellipses and paths which can be styled in sophisticated ways using appropriate attributes.

SVG natively supports affine transformations that can be applied to individual elements to control how they are rendered. Elements can be grouped into hierarchies with different transformations applied to different elements, in order to create intricate arrangements of the elements. See `http://www.carto.net/svg/samples/` for some nicely presented examples of the things that you can do with SVG.

Recommended Strategy

SVG provides a number of built-in features that should make handling the concepts of *HPL+* easier than working with only a format that provided little more than a basic coordinate system. The two most important ideas of SVG that you will probably want to rely upon are the group element (`<g>` tag) and the matrix form of specifying transformations.

In *HPL+* painters are a bit like functions that are awaiting a (painter) frame to be invoked upon. That frame gives meaning to the `frame` and `subframe` references of `paint` statements within a compound painter's body. In SVG, the `<g>` element provides a means to capture all the requirements of an *HPL+* painter. Since a `<g>` element can be given its own transform, which will be applied to all of its children, we can think of that transform as defining the current frame that is given to a painter when it is rendered. Each paint command within a compound painter can be translated into an individual `<g>` element, with its own `transform` attribute if that `paint` command has an accompanying `frame` or `subframe` modifier.

The algebra of painter frames in *HPL+* is a type of affine transformation, and so has a perfect equivalent representation as a matrix transformation. To see this, consider the transformation implicit in finding the screen coordinates of a point $\begin{bmatrix} a \\ b \end{bmatrix}$ in a frame $\left[\begin{bmatrix} o_x \\ o_y \end{bmatrix}, \begin{bmatrix} u_x \\ u_y \end{bmatrix}, \begin{bmatrix} v_x \\ v_y \end{bmatrix}\right]$. (The coordinates of the frames origin and basis vectors are provided in "screen" coordinates where the bottom left corner is (0, 0), the bottom right corner is (1, 0) and the top left corner is (0, 1)). The screen coordinates, $\begin{bmatrix} p_x \\ p_y \end{bmatrix}$ of the point $\begin{bmatrix} a \\ b \end{bmatrix}$ is given by

$$\begin{aligned}
\begin{bmatrix} p_x \\ p_y \end{bmatrix} &= a\mathbf{u} + b\mathbf{v} + \begin{bmatrix} o_x \\ o_y \end{bmatrix} \\
&= a\begin{bmatrix} u_x \\ u_y \end{bmatrix} + b\begin{bmatrix} v_x \\ v_y \end{bmatrix} + \begin{bmatrix} o_x \\ o_y \end{bmatrix} \\
&= \begin{bmatrix} au_x + bv_x + o_x \\ au_y + bv_y + o_y \end{bmatrix} \\
&= \begin{bmatrix} u_x & v_x & o_x \\ u_y & v_y & o_y \end{bmatrix} \begin{bmatrix} a \\ b \\ 1 \end{bmatrix}
\end{aligned}$$

In SVG, the `transform` attribute that may be associated with `<g>` elements can be specified as a sequence of operations, separated by spaces. For example, the `<image>` element below has a transformation on it that scales both its $x-$ and $y-$ coordinates by 0.00167 ($= 1/600$):

```
<image id="crest" x="0" y="0" width="600" height="600"
       transform="scale(0.00167, 0.00167)"
       xlink:href="../images/uwicrest.jpg" preserveAspectRatio="none" />
```

Note that by default, the coordinate system in place is determined by the viewbox defined in the `<svg>` tag at the start of the document. In the examples, it has been set to 700 (width) by 800 (height). So, the image is loaded in to occupy a width and height of 600 by 600 of those virtual pixel units. The scaling transform causes the coordinate system of the image to match the range that *HPL+* painters use.

If the value of the `transform` attribute had been `"translate(50, 50) scale(0.00167, 0.00167)"` instead then the image would first have been scaled and then be translated to have its origin at the point (50, 50) in the SVG display area. These transformations are affine transformations, which really just means that straight lines will be mapped to straight lines. Algebraically, this means that they can be represented as a single matrix. If M is a matrix representing one of these transformations, the image of a point (x, y) under the transformation is obtained by computing the matrix-vector product

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = M \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Moreover, composing two transformations is equivalent to taking a matrix product of the corresponding matrices (earliest applied transformation's matrix to the right) which means that a sequence of transformations can also be represented by a single matrix. For example, the matrix equivalent to the translate, then scale sequence given at the start of the paragraph would be:

$$\begin{bmatrix} 1 & 0 & 50 \\ 0 & 1 & 50 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.00167 & 0 & 0 \\ 0 & 0.00167 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.00167 & 0 & 50 \\ 0 & 0.00167 & 50 \\ 0 & 0 & 1 \end{bmatrix}$$

As a "convenience" SVG allows the transform on an element to also be specified directly as a matrix, using the `matrix` operation. It takes 6 arguments, and the form `"matrix(a1, a2, b1, b2, c1, c2)"` corresponds to the matrix:

$$\begin{bmatrix} a1 & b1 & c1 \\ a2 & b2 & c2 \\ 0 & 0 & 1 \end{bmatrix}$$

So `transform="translate(50, 50) scale(0.00167, 0.00167)"` could also be written as `transform="matrix(0.00167, 0, 0, 0.00167, 50, 50)"`.

In *HPL+* terms, the upshot of this is that rendering in the subframe $\left[\begin{bmatrix} o_x \\ o_y \end{bmatrix}, \begin{bmatrix} u_x \\ u_y \end{bmatrix}, \begin{bmatrix} v_x \\ v_y \end{bmatrix}\right]$ is equivalent to transforming with the matrix:

$$\begin{bmatrix} u_x & v_x & o_x \\ u_y & v_y & o_y \\ 0 & 0 & 1 \end{bmatrix}$$

provided that the image coordinates have already been scaled so that its width and height are both 1 in the new coordinate system. For this reason, you should scale the image upon loading, and then scale it back up upon final rendering. The example SVG fragments exhibit this by the transformations that have been declared on the `<image>` elements and on the outermost `<g>` element that is given the identifier `''screen''`.

An Example

Consider the *HPL+* code fragment given in Listing 1:

This code could be compiled to SVG along two very different paradigms:

Script-Based: Create an SVG file with minimal SVG elements to begin with, and rely on scripts to populate the SVG document as necessary.

Listing 1: A simple *HPL+* program that places two images side by side.

```
crest = img-painter("images/uwicrest.jpg")
wasp = img-painter("images/wasp.jpg")

def-painter beside[a](p1, p2):
   paint p1 in subframe((0, 0), a, 1)
   paint p2 in subframe((a, 0), 1-a, 1)
end

paint beside[0.5](crest, wasp)
```

Element-Based: Create an SVG file with all of the elements to be rendered already present in the file. There is almost no dependence on scripts in this approach.

The two approaches could be viewed as two extremes of a spectrum of combinations of SVG elements and script usage. You are free to elect to make your compiler operate at any point on that spectrum. An example of each approach follows.

The script-based approach

Below is an example of the SVG equivalent to the *HPL+* code fragment, which was generated by hand (so please forgive any inconsistencies there may be). Note that a reasonable variant of this approach would have been to load the script from a file, which would make the SVG file itself nearly constant.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="7cm" height="8cm" viewBox="0 0 700 800"
     xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     onload="init(evt)"
     version="1.1">
  <desc>Manual translation of an HPL script that places 2 images beside
  each other
  </desc>
  <!-- ECMAScript generated by translator -->
  <script type="application/ecmascript"> <![CDATA[
    var svgNS = "http://www.w3.org/2000/svg";
    var xlinkNS = "http://www.w3.org/1999/xlink";
    var initialised = false;

    function init(evt) {
      // a safety measure to ensure that we run only once
      if (! initialised) {
         // translated top level commands go here

         // translating: paint beside[0.5](crest, wasp)
         var a1 = lookup("#crest");
         var a2 = lookup("#wasp")
         var t1 = beside(0.5, a1, a2);
         document.getElementById("screen").appendChild(t1);
```

```
        // end of translated code
        initialised = true;
        }
    }

    function lookup(name) {
      var result = document.createElementNS(svgNS, "use");
      result.setAttributeNS(xlinkNS, "href", name);
      return result;
    }

    function beside(a, p1, p2) {
      // setup return painter representation
      var result = document.createElementNS(svgNS, "g");

      // translate: paint p1 in subframe ((0, 0), a, 1)
      var t1 = document.createElementNS(svgNS, "g");
      t1.setAttributeNS(null, "transform",
                        "matrix(" + a + ", 0, " + "0, 1, 0, 0)");
      t1.appendChild(p1);
      result.appendChild(t1);

      // translate: paint p2 in subframe ((a, 0), 1-a, 1)
      var t2 = document.createElementNS(svgNS, "g");
      t2.setAttributeNS(null, "transform",
                        "matrix(" + (1-a) + ", 0, " + "0, 1, " + a + ", 0)");
      t2.appendChild(p2);
      result.appendChild(t2);
      return result;
    }
  ]]> </script>

  <defs>
    <image id="crest" x="0" y="0" width="600" height="600"
           transform="scale(0.00167, 0.00167)"
           xlink:href="../images/uwicrest.jpg" preserveAspectRatio="none" />
    <image id="wasp" x="0" y="0" width="600" height="600"
           transform="scale(0.00167, 0.00167)"
           xlink:href="../images/wasp.jpg" preserveAspectRatio="none" />
  </defs>

  <g id="screen" x="0" y="0" width="100%" height="100%"
     transform="translate(50, 50) scale(600, 600)" >
    <rect onclick="init(evt)" x="0" y="0" width="1.0" height="1.0"
          stroke="blue" stroke-width="0.01"/>
  </g>
  <text x="300" y="680" font-family="Verdana" font-size="30"
        text-anchor="middle" >
    Click in the box if it is blank
  </text>
</svg>
```

The element-based approach

An alternative approach is to generate the SVG group elements (`<g>` tag) that represent fragments of images directly into the final SVG document. This may be alright for *HPL+* programs that produce

an image directly, but would probably not be very useful for *HPL+* libraries that contained only functions. The output of this approach (again hand-compiled) is shown below. Note that it is almost as if an interpreter had written to an SVG format, instead of to the screen.

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="7cm" height="8cm" viewBox="0 0 700 800"
     xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     version="1.1">
  <desc>Manual translation of an HPL script that places 2 images beside
  each other
  </desc>
  <!-- ECMAScript generated by translator -->
  <script type="application/ecmascript"> <![CDATA[
    var svgNS = "http://www.w3.org/2000/svg";
    var xlinkNS = "http://www.w3.org/1999/xlink";
    var initialised = false;

    function init(evt) {
      // a safety measure to ensure that we run only once
      if (! initialised) {
         // translated top level commands go here

         // end of translated code
         initialised = true;
         }
    }
  ]]> </script>

  <defs>
    <image id="crest" x="0" y="0" width="600" height="600"
           transform="scale(0.00167, 0.00167)"
           xlink:href="../images/uwicrest.jpg" preserveAspectRatio="none" />
    <image id="wasp" x="0" y="0" width="600" height="600"
           transform="scale(0.00167, 0.00167)"
           xlink:href="../images/wasp.jpg" preserveAspectRatio="none" />
  </defs>

  <g id="screen" x="0" y="0" width="100%" height="100%"
     transform="translate(50, 50) scale(600, 600)" >
    <rect onclick="init(evt)" x="0" y="0" width="1.0" height="1.0"
          stroke="blue" stroke-width="0.01"/>
    <g>
      <g transform="matrix(0.5, 0, 0, 1, 0, 0)">
        <use xlink:href="#crest"/>
      </g>
      <g transform="matrix(0.5, 0, 0, 1, 0.5, 0)">
        <use xlink:href="#wasp"/>
      </g>
    </g>
  </g>
  <text x="300" y="680" font-family="Verdana" font-size="30"
        text-anchor="middle" >
    Click in the box if it is blank
  </text>
</svg>
```