Interpreter of SVGen
Homework Assignment 2

Rostislav Horčík

April 16, 2022

1 Introduction

This homework assignment aims to implement an interpreter of a simple programming language which I call SVGen. This language allows writing programs generating SVG images. Thus the interpreter evaluates a given program and returns a string whose content is a valid SVG image. SVG is a XML-based vector image format (for details see `https://en.wikipedia.org/wiki/Scalable_Vector_Graphics`). Our SVGen programs support only a fragment of the SVG specification to keep the assignment simple.

The interpreter should be implemented in Racket. **All your code is required to be in a single file called `hw2.rkt`.** Since Racket allows to manipulate any tree structure as data, your interpreter does not have to parse a text file with a source code, but you will be given an abstract syntax tree (AST) directly represented as a list consisting of the language primitives or other nested lists. An example of a simple SVGen program is shown in Figure 1. This program recursively generates circles with smaller and smaller radii. Once the radius is too small, the program stops. Evaluating the expression `'(recur-circ 200 200 100)` returns a string containing an SVG image depicted in Figure 2.

```
'((define (recur-circ x y r)
    (circle x y r "fill:lightgreen;opacity:0.5;stroke:black;stroke-width:2")
      (when (> r 15)
        (recur-circ (+ x r) y (floor (/ r 2)))
        (recur-circ (- x r) y (floor (/ r 2)))
        (recur-circ x (+ y r) (floor (/ r 2)))
        (recur-circ x (- y r) (floor (/ r 2))))))
```

Figure 1: A simple SVGen program generating recursively circles.

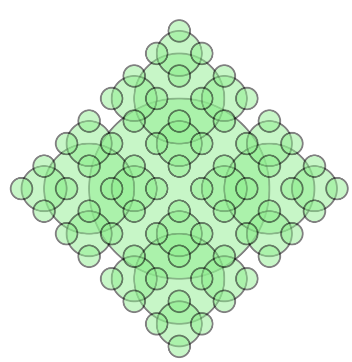

Figure 2: An example SVG image generate by an SVGen program.

2 Interpreter specification

Your task is to implement a function

```
(execute width height prg expr)
```

where `width` and `height` is a width and height of the SVG image respectively. The argument `prg` is an SVGen program consisting of function definitions and `expr` is an expression to be evaluated (typically, it is a function call of a function defined in `prg`). For example, if $width = 500$ and $height = 400$, the `execute` function returns a string

```
<svg width="500" height="400">...content...</svg>
```

where the `...content...` is the result of the interpreter by evaluating `expr` using the function definitions in `prg`. To simplify the task, we split the function definitions part in `prg` from the expression `expr` whose evaluation gives the content of the SVG-tag. So you can process the function definitions `prg` in advance and create a global environment in which the expression `expr` is evaluated afterward. For example the result of the program `prg` in Figure 1 depicted in Figure 2 was returned by the call

```
(execute 400 400 prg '(recur-circ 200 200 100))
```

The function `execute` has to be exported from your file `hw2.rkt`! Thus your file has to contain `(provide execute)`.

3 SVGen syntax and semantics

Now we define the syntax and semantics of SVGen programs.

3.1 Syntax

The syntax of SVGen is specified by a grammar shown in Figure 3. A grammar consists of rules of the form `LHS -> RHS` where `LHS` is a non-terminal symbol and `RHS` is either a sequence of symbols or several sequences separated by the pipe |. The rule states that the non-terminal symbol `LHS` can be rewritten into one of the sequences of symbols separated by |. To define arbitrarily long sequences, we use the formal language operators $\star$ and +. For a symbol `<symbol>` the notation `<symbol>*` (resp. `<symbol>+`) stands for any finite (resp. nonempty) sequence of `<symbol>`s separated by spaces.

We will comment on its parts. An SVGen program is a nonempty list of function definitions as specified by the rule `<program> -> (<definition>+)`. Each function definition is the form `(define (<id> <id>*) <expression>+)`. So it is the same syntax as the function definition in Racket. The first identifier `<id>` is the name of the function and the sequence `<id>*` represents its arguments. The body of the function definition consists of nonempty seqeuence of expressions. Note that there can be no nested definitions.

Each expression is either an application or if-expression or when-expression. The if-expression `(if <bool-exp> <expression> <expression>)` has the same syntax as if-expressions in Racket, i.e., the condition `<bool-exp>` is followed by a then-expression which is followed by an else-expression. The when-expression `(when <bool-exp> <expression>+)` contains a condition `<bool-exp>` followed by a nonempty sequence of expressions. The conditions `<bool-exp>` are of the form `<bool-exp> -> (<bool-op> <num-exp>*)`, where `<bool-op>` is one of `=, <, >` followed by numeric expressions.

The function application `<application>` represents a function call. Either it is a call of an SVG-primitive function (i.e., one of `circle, rect, line`) or a defined function call. Each argument `<arg>` of a function call can be a string or a numeric expression. The numeric expressions `<num-exp>` have the same syntax as in Racket. They are built up from variables, numbers, and functions `+, -, *, /, floor, cos, sin`.

```
<program> -> (<definition>+)

<definition> -> (define (<id> <id>*) <expression>+)

<expression> -> <application>
              | (if <bool-exp> <expression> <expression>)
              | (when <bool-exp> <expression>+)

<application> -> (<svg-op> <arg>*)
               | (<id> <arg>*)

<bool-exp> -> (<bool-op> <num-exp>*)

<arg> -> <string>
       | <num-exp>

<num-exp> -> <num>
           | <id>
           | (<num-op> <num-exp>*)

<string> -> any Racket string, e.g. "fill:blue"
<num> -> any Racket number, e.g. 3.14
<id> -> any Racket symbol, e.g. x
<num-op> -> + | - | * | / | floor | cos | sin
<svg-op> -> circle | rect | line
<bool-op> -> = | < | >
```

Figure 3: The grammar of the programming language SVGen.

3.2 Semantics

The evaluation of an expression with respect to an SVGen program returns a string representing a valid SVG image. The output string is generated by the function calls of SVG-primitives which generate corresponding SVG-tags. The semantics of the SVG-primitives is defined as follows:

- The `(circle x y r style)` is evaluated to the SVG-tag `<circle>`, where x, y are coordinates of its origin, r is its radius, and *style* is a string of style options. For example,

```
(circle 50 40 20 "fill:blue")
  => <circle cx="50" cy="40" r="20" style="fill:blue"/>
```

- The `(rect x y width height style)` is evaluated to the SVG-tag `<rect>`, where x, y are coordinates of its origin, *width*, *height* is its width and height respectively, and *style* is a string of style options. For example,

```
(rect 10 20 30 40 "fill:blue")
  => <rect x="10" y="20" width="30" height="40" style="fill:blue"/>
```

- The `(line x1 y1 x2 y2 style)` is evaluated to the SVG-tag `<line>`, where $x1, y1$ are coordinates of its origin, $x2, y2$ coordinates of the final point, and *style* is a string of style options. For example,

```
(line 10 20 30 40 "stroke:black;stroke-width:5")
  => <line x1="10" y1="20" x2="30" y2="40" style="stroke:black;stroke-width:5"/>
```

The rest of the SVGen semantics is quite straightforward following the Racket semantics. The interpretation of the conditional expression `(when <bool-exp> <expression>+)` evaluates the nonempty sequence of expressions only if the condition `<bool-exp>` is evaluated to true. If the condition `<bool-exp>` is evaluated to false, the when-expression returns the empty string `""`.

Variables can occur in function definitions. The only variables which might occur in a function definition are variables among its parameters. For instance, the body of `(define (f x y) <body>)`

may contain only variables `x,y`. The variables are interpreted based on the values obtained from the function call.

4 Further hints

Try to make your solution structured by splitting your code into several independent pieces and design covering test cases for all the pieces. For example, I split my solution into the following parts:

1. Functions generating SVG-tags
2. Environment functions
3. Syntax functions
4. Evaluator functions

4.1 Functions generating SVG-tags

This part is quite straightforward. You can devise a clever solution using higher-order functions. It is convenient to use the Racket function `format` to create a particular string. For example,

```
(format "<svg width=\"~a\" height=\"~a\">" 200 100)
  => "<svg width=\"200\" height=\"100\">"
```

Note the escape backslash character allowing to enter double quotes. The `format` function also converts numerical values into a string automatically.

4.2 Environment functions

To evaluate an application of a defined function, the interpreter has to know all the function definitions from `prg` and all the values to be bound to the function arguments. You need to design a data structure capturing these data. I call it an environment. The function definitions can be processed in advance because the interpreter gets them separately. Thus it remains constant during the evaluation of a given expression.

The second part of the environment is more dynamic. Once you need to evaluate a function call `(f e1 e2 ...)`, you have to first evaluate the expressions `e1 e2 ...` obtaining some values $v_1, v_2, \ldots$, then you can create a new environment whose second part is created based on the values $v_1, v_2, \ldots$, and finally, you can evaluate the body of `f` in this new environment.

Note that SVGen has no bindings scopes. The only variables are just parameters in the function definitions. Consequently, the second part of the environment is fully determined by a function call. Thus there is no need to extend the envinroment by bindings from the outer scope (unlike a Scheme interpreter).

4.3 Syntax functions

In my case, these functions help me to recognize what kind of expression I am dealing with. For example, if it is an if-expression or a function application. Further, it is convenient to define functions decomposing an expression into particular pieces once recognized. For instance, once it is clear that I process an if-expression, I can apply functions extracting the boolean expression, the then-expression, and the else-expression, respectively.

4.4 Evaluator functions

These functions form the core of the interpreter. They should follow the grammar of the language. Roughly speaking, for each rule of the grammar, there is a corresponding function recognizing which of the right-hand side applies (this can be done by the above-mentioned syntax functions). Once it is clear, the expression is decomposed into particular parts. Each part is either a terminal symbol (like a number, a string, a primitive function) or corresponds to a rule in the grammar. Thus each part can be evaluated. If it is a terminal symbol, its evaluation is given by the semantics (e.g., the symbol `'+` stands for the Racket function `+`). If it corresponds to a rule, it can be evaluated recursively by a corresponding evaluator function.

5 Test cases

This section presents a few test cases to show how the interpreter should behave. Similarly, as in the first homework assignment, your `execute` function returns a string. If you test it directly in DrRacket REPL, the REPL displays the string value full of the escape character `\`. To see the result without escape characters, one has to apply function `display` to the result. Such displayed string can then be copied into your clipboard and pasted into any SVG viewer (I use the online editors in `https://www.w3schools.com/graphics/svg_intro.asp`). In the following examples, we will display the resulting SVG format on several lines indented for better readability. However, to simplify your task, your output string does not need to contain any newline characters or whitespace between SVG tags.

1. ```
   (display (execute 400 400 '() '(line 10 20 30 40 "stroke:black;stroke-width:5")))
     => <svg width="400" height="400">
          <line x1="10" y1="20" x2="30" y2="40" style="stroke:black;stroke-width:5"/>
        </svg>
   ```

2. ```
   (display (execute 400 400 '() '(circle 200 200 (floor (/ 200 3)) "fill:red")))
     => <svg width="400" height="400">
          <circle cx="200" cy="200" r="66" style="fill:red"/>
        </svg>
   ```

3. ```
   (define test1
     '((define (start)
         (rect 0 0 100 100 "fill:red")
         (rect 100 0 100 100 "fill:green")
         (rect 200 0 100 100 "fill:blue"))))
   (display (execute 400 400 test1 '(start)))

   => <svg width="400" height="400">
        <rect x="0" y="0" width="100" height="100" style="fill:red"/>
        <rect x="100" y="0" width="100" height="100" style="fill:green"/>
        <rect x="200" y="0" width="100" height="100" style="fill:blue"/>
      </svg>
   ```

4. ```
   (define test2
     '((define (circles x r)
         (when (> r 10)
           (circle x 200 r "fill:white;stroke:green")
           (circles (+ x (floor (/ r 2.0))) (floor (/ r 2)))))))
   (display (execute 400 400 test2 '(circles 200 200)))

   => <svg width="400" height="400">
       <circle cx="200" cy="200" r="200" style="fill:white;stroke:green"/>
       <circle cx="300.0" cy="200" r="100" style="fill:white;stroke:green"/>
       <circle cx="350.0" cy="200" r="50" style="fill:white;stroke:green"/>
       <circle cx="375.0" cy="200" r="25" style="fill:white;stroke:green"/>
       <circle cx="387.0" cy="200" r="12" style="fill:white;stroke:green"/>
      </svg>
   ```

```
(define tree-prg
  '((define (draw x1 y1 x2 y2 len angle)
      (if (> len 30)
          (line x1 y1 x2 y2 "stroke:black;stroke-width:2;opacity:0.9")
          (line x1 y1 x2 y2 "stroke:green;stroke-width:3;opacity:0.9"))
      (when (> len 20)
        (recur-tree x2 y2 (floor (* len 0.7)) angle)
        (recur-tree x2 y2 (floor (* len 0.7)) (+ angle 0.3))
        (recur-tree x2 y2 (floor (* len 0.7)) (- angle 0.6))))
    (define (recur-tree x1 y1 len angle)
      (draw x1
            y1
            (+ x1 (* len (cos angle)))
            (+ y1 (* len (sin angle)))
            len
            angle))))
```

Figure 4: The SVGen program generating a tree.

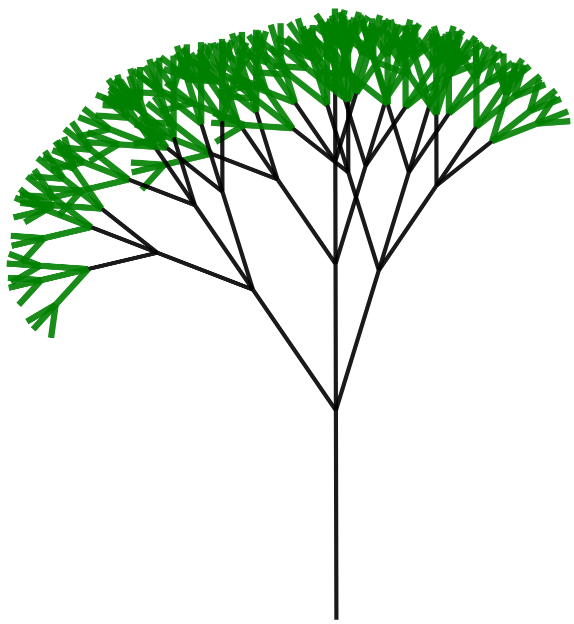

Figure 5: The tree SVG image

Note that x coordinates of the circles (but the first one) are decimal numbers. This follows from their computation by the expression `(+ x (floor (/ r 2.0)))` because `2.0` is treated as a real number. So the result is a real number as well. On the other hand, the radii are computed by `(floor (/ r 2))` returning an integer. For instance, `(/ 25 2)` is evaluated as $12\frac{1}{2}$. Consequently, `floor` returns the integer 12.

5. A more complex example generating recursively a tree is shown in Figure 4. Evaluating this program by

   ```
   (display (execute 400 400 tree-prg '(recur-tree 200 400 100 (* 3.14 1.5))))
   ```

   generates the tree depicted in Figure 5.