National University of Singapore
School of Computing
CS1101S: Programming Methodology
Semester I, 2013/2014

Mission 2
Rune Reading

Start date: 17 August 2013
Due: 22 August 2013, 23:59

Readings:

- Textbook Sections 1.1.1 to 1.1.4

Your instructors are pleased with your performance in the first mission. In this new mission, you are expected to demonstrate your investigative skills. Mastery of the Force is nothing without the ability to think and reason.

You need to open three doors, leading you another step closer towards the inner sanctum of the JFDI temple.

In Lecture 2, we demonstrated how JediScript can be used to generate runes. Now, you get to try your hand at drawing them. You can use the Console tab on the Mission page to experiment with drawing runes. The runes will be drawn in the Display tab. You will also be able to write in the Source tab, save this, and click Run to execute the code.

Alternatively, you have been provided with JediScript template files and an HTML file with a viewport. You can work on your Mission with these. You should complete your tasks within the template files, and you can then drag-and-drop the JediScript file (from an explorer window) onto the viewport (in the HTML file opened in the browser). Refreshing the HTML file will clear the canvas. Once you are satisfied with your answer(s), then you must upload your code into the Code area on the JFDI Academy for Mission 2.

For this mission, JFDI loads the `runes.js` script library. This provides the four runes discussed in class `rcross_bb`, `sail_bb`, `corner_bb`, and `nova_bb`. For example, you can display `rcross_bb` in the viewport with the following command in the Console:

```
show(rcross_bb);
```

Don't forget to clear the viewport when necessary with `clear_all();`. If you are making use of the HTML files outside of JFDI Academy, the HTML files will load the `runes.js` script library, and has the Viewport and runes similar to the JFDI Academy site.

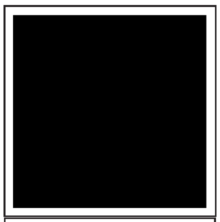
Also defined in `runes.js` are the following functions as discussed in class:

- `stack`
- `quarter_turn_right`
- `flip_vert`
- `stack_frac`
- `flip_horiz`
- `turn_upside_down`

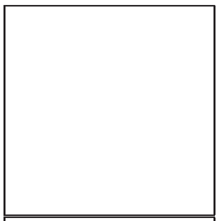
- `quarter_turn_left`
- `beside`
- `make_cross`
- `repeat_pattern`
- `stackn`

In addition to the ones you saw in Lecture, we have also defined a few more basic runes that you can use:

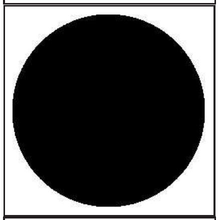
`black_bb`:



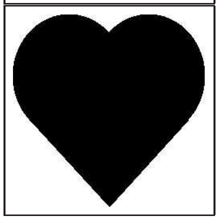
`blank_bb`:



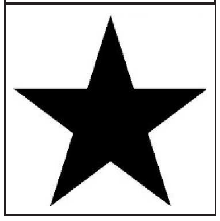
`circle_bb`:



`heart_bb`:



`pentagram_bb`:



In writing rather large, complex programs, one does not often understand (or even see) every single line of code, since such programs are usually written by several people, each in charge of smaller components. The key idea in functional abstraction is that as the programmer, you need not understand how each function works. All you need to know is what each function does and its signature (such as what parameters to pass). More specifically, **you need not read the code for the predefined functions listed above**. You may refer to the lecture slides to understand what arguments each function requires and its corresponding effect. Now we will use these functions as primitives building blocks to draw our own pictures.

This mission consists of **three** tasks.

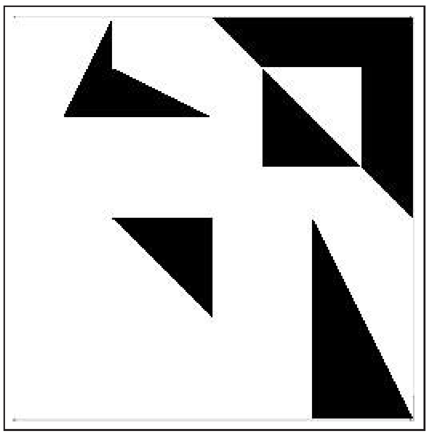
Task 1:

On the first door you find 4 basic runes and one complex rune separated by an empty space. Clearly, your task would be to fill in this space with a descriptive function for the manipulation of the 4 basic runes to create the complex one.

Write a function `mosaic` that takes four runes as arguments and arranges them in a 2×2 square, starting with the top-right corner, going clockwise. In particular, the command

```
show(mosaic(rcross_bb, sail_bb, corner_bb, nova_bb));
```

should draw the following:



Task Files

- lib/list.js
- lib/misc.js
- lib/graphics.js
- lib/runes.js
- mission_2_1.html
- **mission_2_1.js**

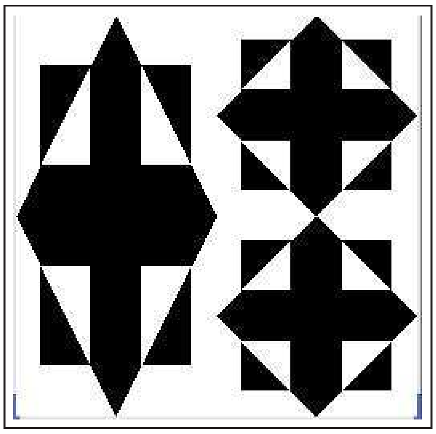
Task 2:

On the second door, you find 2 runes displayed in a similar fashion. The only difference would be that the complex rune now exhibits a different layout.

Write a function `simple_fractal` that takes as argument a rune and returns a rune consisting of the original rune and a pair of runes stacked on top of each other on its right. For example, the following command:

```
show(simple_fractal(make_cross(rcross_bb)));
```

should draw the following:



Task Files

- lib/list.js
- lib/misc.js
- lib/graphics.js
- lib/runes.js
- mission_2_2.html
- **mission_2_2.js**

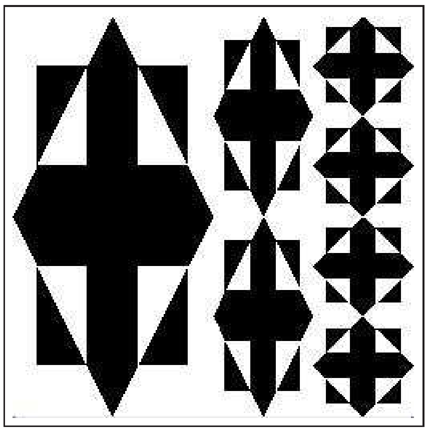
Task 3:

In the third and final door, you behold a similar sight to the previous door albeit with a slight difference. It will be well for you to comprehend the similarities and differences.

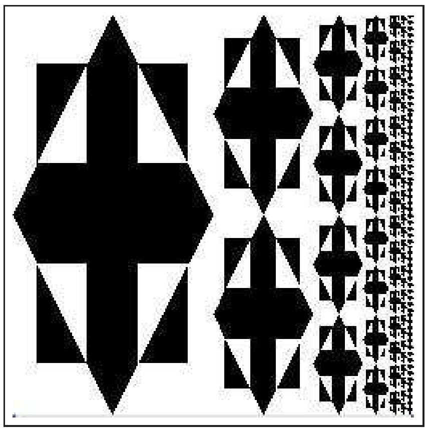
Write a function `fractal` that takes as arguments a rune and an integer $n > 0$. It should generate the rune below by means of a recursive process with the following command:

```
show(fractal(make_cross(rcross_bb), 3));
```

This should draw the following:



To determine that your function is correct for $n > 3$, check that the same command with $n = 7$ draws:



Task Files

- lib/list.js
- lib/misc.js
- lib/graphics.js
- lib/runes.js
- mission_2_3.html
- **mission_2_3.js**

Submission

To submit your work to the Academy, place your code in the Source Code section of the online editor within the mission page, save the code by clicking the Save button, and click the Finalize

Submission button at the bottom of the page. Please ensure the required function from each Task is included in your submission. Note that submission is final and that any mistakes in submission requires extra effort from a tutor or the lecturer himself to fix. IMPORTANT: Make sure youve saved the latest version of your work by clicking the Save button before finalizing your submission!