ECE 421 Assignment 3 Part 1

Winter2019_Group4:

Nathan Klapstein (1449872)

Tony Qian (1396109)

Thomas Lorincz (1461567)

Zach Drever (1446384)

Table of Contents

Design Questions

1. **What is your definition of an object?**

   Objects are instances of classes. In object oriented programming there is one root super class which defines a set of methods which every class must define. These methods typically define methods for null checking, object equality, object hashing, accessing object type or displaying the object.

   Classes are definitions for a data format and available procedures. They may also contain data and procedures (known as class methods) themselves, i.e. classes contain the data members and member functions

2. **What strategies should be deployed in terms of accepting input (i.e. the large number of objects.)?**

   We plan to support using both Arrays and Enumerable objects (since we will only access the *.count, *.sort, *.each_slice methods/class variables). Thus, using enumerable objects it should be possible to properly lazily evaluate and avoid potential issues caused by extreme input size.

3. **Is your sort generic? What sorting criterion does your system use? Is this criterion flexible; i.e. changeable by the user at the point of execution? What limitations have you placed upon legal sorting criteria? To make it reusable to other people, how should we include it into the Ruby Class hierarchy?**

   Sort should be generic similar to how the Ruby's Enumerable and Array classes implement the *.sort method. Ideally it should be able to sort any Ruby object as the Ruby Objects class specifies methods for 'less than', 'greater than' and 'equal'. Objects should keep track of what criterion they should be sorted on (ie. what data fields to compare) and they can simply be ordered using a comparator that returns 1, 0 or -1 depending on the data field to compare. For example, a complex object with many data fields may not be trivial to compare, and the data may be of separate types (strings, integers). In this case, a comparator that can compare all data fields might be necessary to order the objects appropriately.

4. **In reality we have a uniprocessor system, describe "equationally" what is happening to your solution as you increase the concurrency (e.g. produce a regression model (remember regression from Statistics) of processing time against the number of threads. Your solution can be modeled as a program which: (1) has a component which produces threads; and (2) a set of threads which undertake the same (small) task. This is in essence the basis of stress testing (discussed in ECE 322)**

CRuby, the most common type of Ruby implements a Global Interpreter Lock which will not allow thread parallelization. JRuby is a Ruby implementation that will utilize Java Threads to achieve true parallelization.

The speedup from parallelizing an algorithm is achieved by dividing the work among threads which can each perform some portion of the computation of the algorithm. However, spawning threads creates some computational overhead before you can begin parallelizing. Additionally, unlike separate processes, threads share variables which can cause unpredictable results if the reading and writing of variables is not synchronized across threads. There is typically overhead caused by synchronization of threads as well as context switching on the processors.

Looking at Amdahl's law doubling the number of processing elements (threads in our case) should halve the runtime, and doubling it a second time should again halve the runtime. However, very few parallel algorithms achieve optimal speedup. Most of them have a near-linear speedup for small numbers of processing elements, which flattens out into a constant value for large numbers of processing elements.

The potential speedup of an algorithm on a parallel computing platform is given by Amdahl's law[15]

S_latency(s) = 1 / (1 - p + p/s)
Where:
S_latency: is the potential speedup in latency of the execution of the whole task;
s: is the speedup in latency of the execution of the parallelizable part of the task;
p: is the percentage of the execution time of t
he whole task concerning the parallelizable part of the task *before parallelization*.

5. **Concurrent systems tend to crash frequently – what approach to exception-handling have you devised? Consider the content of the library at: http://c2.com/cgi/wiki?ExceptionPatterns; which are applicable to this problem? Is Module Errno useful in this problem? What components of the Ruby exception hierarchy are applicable to this problem? Discuss in detail your strategy for exception-handling.**

Pattern 3 nullcatchcase allows for extremely trivial exceptions to be ignored thus reducing exception load at higher levels in the programs

Pattern 2 convert exceptions / nested exceptions -> possibly convert or nest exceptions so that more broad catches can handle a varying group of errors. This could be used to group exceptions that are critical vs trivial and deduplicate the code for handing such cases.

Pattern 1 Abort, retry, ignore can be used to command threads on crash to possibly recover, and if critical failure abort. Or ignore (e.g. other thread crashes shouldn't stop other sorting thread for now.)

Summary: catch and don't raise trivial exceptions, group exceptions in two categories critical and trivial, and handle critical vs trivial exceptions appropriately either retrying, ignoring, or failing where appropriate for that group.

Argument error checking checking if content is sortable is already provided via the native *.sort method and argument error checking for the number of process / thread workers is natively provided by the Parallel gem. Process / thread errors are natively wrapped and managed effectively by the Parallel gem, thus, removing the need of manually managing threads / processes on exceptions. Using the direct exception bubbling already provided is explicit and avoids duplication of error handling already provided via native libraries and gems. Test assertions of this expected exception behavior are provided in /test/tort_test.rb.

6. **What differences exist between thread-based and process-based solutions? How has this impacted the design of your solution?**

Threads based share memory. Threads can directly edit the memory/objects to their other related threads. (e.g. adding to a queue). Compared to processes, they usually have lower memory overhead. Using threads introduces several synchronization problems because of the data they share with other threads. These include race conditions (threads completing out of expected order), invalid memory accessing (memory corruption), and deadlocks (threads consuming resources in such a manner that the system cannot proceed) can all be introduced if programming threads is done recklessly.

Process based are copies of the execution code with separate memory maps. Processes can communicate (usually) via serialized pipes. Interprocess communication usually involves much more overhead compared to cross-thread communication. Synchronization is (usually) more difficult for processes as interprocess control is usually less detailed in most OS environments.
Look at https://cs.uwaterloo.ca/~mashti/cs350-w18/processes.pdf

7. **Do you have any race-condition or task synchronization concerns about your solution? How do we tidy-up a multi-threaded program, if stopped mid-execution?**

Concurrent, thread based solutions which must read and write to the same in memory data structures will likely give rise to race-conditions. The most effective way to deal with these task synchronization concerns is to use some sort of synchronization technique across threads. Common inter-thread synchronization objects are mutexes, semaphores or read-write locks. These objects can be used to synchronize the reading and writing from shared memory locations (the critical sections), so that the outcome of reads and writes across threads are not subject to race conditions.

The way in which a programmer should tidy up a multi-threaded program which is stopped mid execution is entirely dependant on the nature of the program, and what causes the interruption. For some programs, it may be possible to gracefully exit with some sensible return

value. However, for interrupting exceptions, SIGINT or KILL signals, the program should simply stop execution and cleanup it's memory.

8. **As discussed in CMPUT 301:**
   a. **What is configuration management?**

      Configuration management (CM) is the practice of handling changes systematically so that a system maintains its integrity over time. CM implements the policies, procedures, techniques, and tools that manage, evaluate proposed changes, track the status of changes, and maintain an inventory of system and support documents as the system changes. CM programs and plans provide technical and administrative direction to the development and implementation of the procedures, functions, services, tools, processes, and resources required to successfully develop and support a complex system.
      https://en.m.wikipedia.org/wiki/Configuration_management

   b. **What is version control?**

      Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. It ensures that a historical timeline of a file(s) is created as the file undergoes modifications. Examples of version control systems are Git, Mercurial, Subversion, etc.

   c. **Are you using either concept? Describe your process and any tool support what you utilize – illustrate your process with regard to Assignments 1 and 2**

      For the usage of configuration management:

      GitHub, Travis-CI, and CodeCov are used within our group to form a trifecta that stores our projects code (GitHub), documents issues and project goals (GitHub), validates changes applied (Travis-CI), and provides overviews on the validations on the project (Travis-CI and CodeCov). All of these components are relatively hands-free (low configuration, hosting is remote outside of our control, and are highly automated), thus, saving us time in developing our projects.

      For usage of version control:

      We are using the version control software git to assist in developing all assignments in ECE 421. Developers of our group use git to easily branch code off the original master source branch and develop various features without road-blocking each other. Changes to the source code are documented via commit messages and provide an effective iterative timeline to the mutation of the development branch. After the development branch finishes implementing the feature it set out to develop, the branch is used to start a pull request, which is reviewed by other group peers and adjusted as needed. When a pull request is up to the groups standards it is merged into the projects master branch, thus, adding the feature to the primary version of the source code.

9. What is JRuby?

A Java/JVM implementation of Ruby for increased speed it utilizes just in time compilation for speedup.

One important note for this assignment is that JRuby does not have the Global Interpreter Lock (GIL) that is enabled in normal C Ruby. Thus, JRuby can have multiple threads execute at the same time. As such, JRuby can potentially result in a speedup in a script utilizing multi-threading when compared to the standard C Ruby execution of the same script. Since Assignment 3 utilizes multi-threading it may be beneficial to experiment with JRuby for performance gains.

10. Briefly Explain:

a. What is refactoring (as discussed in ECE 325 / CMPUT 301)?

The main idea is to change a software system such that the external behavior (behavior observed by the outside software user) does not change, but, the internal structure of the software is improved. This is mainly done to assist in future development and maintenance efforts as refactored code should be easier to read, test, and extend. The refactoring process should detect smells (bad coding practices) within software and remedy them with an appropriate refactoring pattern.

b. Are you using refactoring in your development process? Justify your answer?

Yes, we have used refactoring. See below:

c. If "yes", give examples, minimum of 2, of the refactoring "patterns" that you used in Assignment 1

Example 1:

In **Assignment 1** the dictionary of keys (DOK) implementation `DOKMatrix` within `dok_matrix.rb` implemented the `DOKMatrix.[]=(row, col, value)` method which set a value within the DOK matrix. Two major restrictions on invocation of the `DOKMatrix.[]=(row, col, value)` method are present:

Setting values on negative indexes of the matrix (as matrices cannot have negative indexes under normal conditions).

Below is an example of an invalid indexing via the `DOKMatrix.[]=(row, col, value)` method:

```
dok_matrix[-1, -1] = 0
```

Setting a non-`nil` or non-`Numeric` value within a numerical matrix.

Below is an example of an invalid assignment via the `DOKMatrix.[]=(row, col, value)` method:

```
dok_matrix[row, col] = "invalid"
```

These two variants of invalid invocation of the `DOKMatrix.[]=(row, col, value)` method should never be allowed to execute without error. And should not be invoked under normal operating conditions.

One solution to ensure that all invocations `DOKMatrix.[]=(row, col, value)` are valid is by adding conditional guards before all calls of the `DOKMatrix.[]=(row, col, value)` method, preventing invalid invocation on failing the guard's conditionals. However, this solution would be duplicative, prone to error, and hard to maintain.

Below is an example of using an conditional guard before invoking the `DOKMatrix.[]=(row, col, value)` method:

```
if row >= 0 && col >= 0 || value.is_a?(Numeric) ||value.nil?
      # code to handle invalid access
else
      dok_matrix[row, col] = value
end
```

While this ensures that the invalid invocations of the `DOKMatrix.[]=(row, col, value)` method are avoided. Doing this for every call of the `DOKMatrix.[]=(row, col, value)` method can quickly become repetitive.

To resolve duplication and maintenance issues present within the first solution, the "Introduce Assertion" refactoring pattern was applied (in our case specific exception raising was used instead of raw assertions for better debugging context). This moves the conditional guards that were placed before calls of the `DOKMatrix.[]=(row, col, value)` method into the `DOKMatrix.[]=(row, col, value)` itself. Thus, removing the duplication, and maintenance issues posed by the first (above) solution. Whilst ensuring execution raises / halts on a invalid invocation of the `DOKMatrix.[]=(row, col, value)` method, thus, effectively applying the invocation restrictions of the `DOKMatrix.[]=(row, col, value)` method as defined above in (1. and 2.).

Below is the implementation of the "Introduce Assertion" refactoring pattern that was applied into the `DOKMatrix.[]=(row, col, value)` method within `dok_matrix.rb`:

```
def []=(row, col, val)
      raise(ArgumentError) unless row >= 0 && col >= 0
      raise(TypeError) unless val.is_a?(Numeric)|| val.nil?
      # other []= method code…
end
```

Example 2:

In **Assignment 2** functionality creating a temporary file was duplicated among tests within `basic_shell_test.rb`. This proved to be hard to maintain between tests, and disorganized.

Below is an example of the original temporary file generation code for tests that was duplicated across tests is shown below:

```
def test_cat
      tempfile = Tempfile.new("test_file")
      tfd = tempfile.open
      tfd.write("test content")
      tfd.close
      # test specific code…
end

def test_cat_multiple
      test_file1 = Tempfile.new("test_file1")
      tfd1 = tempfile1.open
      tfd1.write("tesst content 1")
      tfd1.close

      test_file2 = Tempfile.new("test_file2")
      tfd2 = tempfile2.open
      tfd2.write("test content 2")
      tfd2.close
      # test specific code...
end
```

This was resolved by applying the "extract-method" refactoring pattern. Extracting the temporary file creation functionality into function called `create_tempfile_test_file` within `helper.rb` and using it within tests, thus, lowering code duplication.

Below is the definition of `create_tempfile_test_file` within `helper.rb`:

```
def create_tempfile_test_file(name, content)
      tempfile = Tempfile.new(name)
      tfd = tempfile.open
      tfd.write(content)
      tfd.close
      assert_true(File.exist?(tempfile.path.to_s))
      tempfile
end
```

Below is the usage of `create_tempfile_test_file` within `basic_shell_test.rb` removing the code duplication mentioned before:

```
def test_cat
      test_file = create_tempfile_test_file('test_file', 'test
      content')
      # test specific code…
end

def test_cat_multiple
      test_file1 = create_tempfile_test_file('test_file1', 'test
      content 1')
      test_file2 = create_tempfile_test_file('test_file2', 'test
      content 2')
      # test specific code…
end
```

11.

a. Describe in Detail what the following Ruby code does.

```
module Enumerable
      def sloth
            Sloth.new(self)
      end
end

class Sloth< Enumerator
      def initialize(postbox)
            super() do |action|
                  begin
                        postbox.each do |val|
                              if block_given?
```

```
                                    yield(action, val)
                               else
                                    action << val
                               end
                          end
                     rescue StopIteration
                     end
               end
          end

          def map(&block)
               Sloth.new(self) do |action, val|
                    action << block.call(val)
               end
          end

          def take(n)
               taken = 0
               Sloth.new(self) do |action, val|
                    if taken < n
                         action << val
                         taken += 1
                    else
                         raise StopIteration
                    end
               end
          end
     end
```

This piece of code defines both a module (Enumerable), and a class (Sloth). Sloth extends Enumerator which is a standard library class in Ruby designed to iterate objects. Enumerator objects typically yield elements one by one, so that some operation can be applied to each object in a collection. Sloth is a class which can be used to chain commands to a previous enumerator. The Enumerable module defined here has one method which can be called on an Enumerator to return a new Sloth object. The Sloth object can then chain subsequent calls to ‘map’ or ‘take’. Both methods defined in Sloth (‘map’ and ‘take’) require to be passed and return new Enumerators. The ‘map’ object takes the previous Sloth object and applies the new block to the object. The ‘take’ method simple returns a Sloth object that contains only the first *n* objects in the previous Sloth Enumerator. Take will raise a StopIteration error after *n* objects have been returned.

b. Explain in detail what happen when we execute the following code block

```
p 1.upto(Float::INFINITY).sloth
   .map { |x| x*x }
   .map { |x| x+1 }
   .take(5)
   .to_a
#=  [2, 5, 10, 17, 26]
```

This block of code takes an Enumerator that will return all of the possible values from 1 -> Float::INFINITY, turns it into a Sloth Enumerator. The Sloth Enumerator then applies two functions to each element in the original Enumerator: squaring, and then addition by one. Because the 'take' operation Raises a StopIteration error, and Enumerators return objects lazily, even though the original Enumerator should enumerate from 1 -> Float::INFINITY, only 5 Enumerations will be completed before the error is raised, stopping the generation of values. 'to_a' is then used to cast the resulting Enumerator to an array object.

12.

a. Consider the following Java code segment, what design pattern is being implemented?

```
public class DesignPattern {
      private static DesignPattern pattern;
      private DesignPattern(){}
      public static DesignPattern getPattern(){
            if(pattern == null){
                  pattern = new DesignPattern();
      }
return pattern;
      }
}
```

This code segment is an example of the singleton pattern. The singleton pattern aims to have only a single instance of a singleton class that is used statically across the rest of the codebase (that has the privilege to do so). When it comes to initializing a singleton instance, there are a few ways to accomplish the same behaviour. The above example uses lazy initialization.

b. The following code segment seeks to improve the above pattern. Fully explain what improvements are being made. Why do you think the code has two if statements?

```
public static NewDesignPattern getNewPattern(){
      if(pattern == null){
            synchronized (NewDesignPattern.class) {
```

```
                    if(pattern == null){
                        pattern = new NewDesignPattern();
                    }
                }
            }
            return pattern;
    }
```

The above code example uses lazy initialization but ensures that there is no race condition among multiple threads that might be initializing the singleton. This is accomplished using the “synchronized” keyword which only allows a single thread to initialize the code block within its scope (similar to using a semaphore).

The first null check ensures that threads that are late to initialize will not go into the synchronized block because the object is already initialized. The second null check ensures that any blocking threads that are inside of the synchronized block will not instantiate the object (again) once the executing thread finishes.

c. How would you implement this improved pattern in Ruby?

Ruby provides a singleton module that can be included in any class. Once included, the module will ensure that there is only one instance of the class. It can be referenced by accessing DesignPattern.instance.

```
include 'singleton'

class DesignPattern
    include Singleton
    # ...
end
```

d. What is the difference in thread-models, between Java and Ruby?

MRI/CRuby (as we are using in this class) implements threads as Green Threads within its interpreter. Green Threads are user-level threads that cannot take advantage of multiple cores (some exceptions apply). Green Threads are allocated on the heap as opposed to having the OS creating new stack space for the thread. Many of the advantages of threads are present when using green threads but parallelism is not.

Java uses JVM Threads (Native Threads). Native Threads are traditional, OS-based threads that are allocated with their own stack space. These threads can take advantage of multiple cores (some exceptions may apply).

e. What is the difference in thread-models, between Ruby and JRuby?

JRuby implements threads similarly to Java (they both use JVM implementation of threads). Thus, the same differences highlighted between Java and Ruby above apply to the answer of this question as well.

13. Consider the following Java code segment:

Shares.java

```
public static final List<String> symbols=
Arrays.asList("IBM","AAPL","AMZN","CSCO","SNE","GOOG","MSFT","ORCL","
FB","VRSN");
```

GoogleFinance.java

```
public static BigDecimal getPrice(final String symbol) {
      try {
            URL url=new
URL("https://finance.google.ca/finance?q="+symbol);
            final BufferedReader reader= new BufferedReader(new
InputStreamReader(url.openStream()));
            final String
data=reader.lines().skip(170).findFirst().get();
            final String[] dataItems=data.split(">");
            return new BigDecimal(
NumberFormat.getNumberInstance().parse(dataItems[dataItems.length1].s
plit("<")[0]).toString());
      } catch (Exception ex) {
            throw new RuntimeException(ex);
      }
}
```

ShareInfo.java

```
public class ShareInfo {
      public final String symbol;
      public final BigDecimal price;

public ShareInfo (final String theSymbol, final BigDecimal thePrice)
{
      symbol = theSymbol;
      price = thePrice;
      }
      public String toString() {
            return String.format("symbol: %s price: %g", symbol,
price);
      }
}
```

ShareUtil.java

```
public class ShareUtil {
     public static ShareInfo getPrice(final String symbol) {
     return new ShareInfo (symbol, GoogleFinance.getPrice(symbol));
     }
public static Predicate<ShareInfo> isPriceLessThan(final int price) {
          return shareInfo ->
shareInfo.price.compareTo(BigDecimal.valueOf(price)) < 0;
     }
     public static ShareInfo pickHigh(final ShareInfo share1, final
ShareInfo share2) {
          return share1.price.compareTo(share2.price) > 0 ? share1:
share2;
     }
 }
```

PickShareImperative.java

```
ShareInfo highPriced = new ShareInfo ("", BigDecimal.ZERO);
final Predicate isPriceLessThan500 = ShareUtil.isPriceLessThan(500);
for(String symbol : Shares.symbols) {
ShareInfo shareInfo = ShareUtil.getPrice(symbol);
     if(isPriceLessThan500.test(shareInfo)) highPriced =
ShareUtil.pickHigh(highPriced, shareInfo);
}
System.out.println("High priced under $500 is " + highPriced);
```

The previous code simply compares some shares' prices and returns the stock with the highest price whose value is less than 500$. However, the code uses an imperative style. As you can see, the code uses mutating variables. Furthermore, if we want to change the logic a little (i.e. pick the share with the highest price under $1,000), we will have to modify this code. This makes the code not reusable.

a. **Rewrite the previous code in a functional style. You should create a class named PickShareFunctional in which you will define one method called findHighPrices.**

   To help you, this is how you should call findHighPrices method:
   `findHighPriced(Shares.symbols.stream());`

   As you can see, converting symbols from List to a Stream of symbols will allow you to use the JDK specialised functional-style methods (i.e map, reduce, and filter). Hint: In this method, you should do the following steps:

   1- Create a list of ShareInfo filled with the price for each of the symbols in Shares

2- Trim down this list to a list of shares whose prices under $500
3- Return the highest-priced share.

```
    public static void findHighPriced(Stream<String> shares) {
   System.out.println("High priced under $500 is "+shares.map(
           s -> ShareUtil.getPrice(s)
   ).filter(
           i -> ShareUtil.isPriceLessThan(500).test(i)
   ).reduce(
           (shareInfo1, shareInfo2) -> ShareUtil.pickHigh(shareInfo1,
shareInfo2)
   ));
}
```

b. Calculate the execution time for the findHighPriced method. i.e try to use timing methods in java before your call to findHighPriced methods.

What is the execution time for the findHighPriced method?

This program is unable to run because the network request provided in the GoogleFinance.java continually returns a 403 error. I would estimate that the 10 network requests run sequentially would take a few seconds, plus the processing would take an additional half second.

Out of the three steps explained above in part (a), which step do you think is responsible for most of this time?

The 10 network requests to getStockPrice() would be the most time intensive.

c. Change the way you are calling the findHighPriced method as:
`findHighPriced(Shares.symbols.parallelStream());`

Calculate the execution time as in part (b) again, but now with parallelStream() instead of stream(). Explain why the execution times are different?

Because the network requests can be carried out concurrently rather than blocking while waiting for a response from the google API. The execution time should be reduced to around 1 second.

14. Create a class RLisp which presents Ruby implementation of the Lisp language. Lisp is a family of computer programming languages with a fully parenthesized prefix notation.

The implementation must include declaration of the following functions and special forms:

- **label: bind a value to a name**
- **quote: avoids evaluation of an expression**
- **car: returns the first element in a list**
- **cdr: returns all but the first element in a list**
- **cons: appends elements to the front of a list and returns a new list.**
- **eq: checks if two values are equal.**
- **if: checks if a given expression is true and evaluates one of two branches depending on the result**
- **atom: checks if a value is a base value (i.e. not a list)**
- **lambda: creates a function Here is a sample set of test cases to assist you in constructing your program.**

```
l = RLisp.new
l.eval [:label, :x, 15]
l.eval :x #=> 15
l.eval [:eq, 17, :x] #=> false
l.eval [:eq, 15, :x] #=> true
l.eval [:quote, [7, 10, 12]] #=> [7, 10, 12]
l.eval [:car, [:quote, [7, 10, 12]]] #=> 7
l.eval [:cdr, [:quote, [7, 10, 12]]] #=> [10,12]
l.eval [:cons, 5, [:quote, [7, 10, 12]]] #=> [5, 7, 10, 12]
l.eval [:if, [:eq, 5, 7], 6, 7] #=> 7
l.eval [:atom, [:quote, [7, 10, 12]]] #=> false
l.eval [:label, :second, [:quote, [:lambda, [:x],   [:car,
[:cdr, :x]]]]]
l.eval [:second, [:quote, [7, 10, 12]]] #=> 10

# Credit for RLisp to
https://github.com/fogus/ulithp/blob/master/src/lithp.rb

class RLisp
   def initialize()
       @context = {
           :label => proc { |(name,val)| @context[name] = eval(val,
@context) },
           :car   => lambda { |(list), _| list.first },
           :cdr   => lambda { |(list), _| list.drop 1 },
           :cons  => lambda { |(e,cell), _| [e] + cell },
           :eq    => lambda { |(l,r), ctx| eval(l, ctx) == eval(r, ctx)
},
           :if    => proc { |(cond, thn, els), ctx| eval(cond, ctx) ?
```

```
eval(thn, ctx) : eval(els, ctx) },
           :atom  => lambda { |(sexpr), _| (sexpr.is_a? Symbol) or
(sexpr.is_a? Numeric) },
           :quote => proc { |sexpr| sexpr[0] }
       }
   end

   def eval sexpr, context=@context
     return context[sexpr] || sexpr if context[:atom].call [sexpr],
context
     function, *args = sexpr
     args = args.map {
         |arg| eval(arg, context)
       } if context[function].is_a?(Array) ||
(context[function].respond_to?(:lambda?) && context[function].lambda?)
     return context[function].call(args, context) if
context[function].respond_to? :call
     eval context[function][2],
context.merge(Hash[*(context[function][1].zip args).flatten(1)])
   end
 end
```

The RLisp class can also be found with our code base in the `lib/` directory. Testing for RLisp can be found in the `test/` directory.

References

Project source code repository:
https://github.com/ECE421/tort

Tutorialspoint Ruby tutorials that notes what objects are defined as in ruby:
https://www.tutorialspoint.com/ruby/ruby_classes.htm

Introduce assertion refactoring pattern:
https://refactoring.guru/introduce-assertion

Extract-method refactoring pattern:
https://refactoring.guru/extract-method

Threading implementations in different Ruby implementations:
https://stackoverflow.com/questions/56087/does-ruby-have-real-multithreading