CSCI3180 – Principles of Programming Languages – Spring 2014

Assignment 3: Perl, Dynamic Scoping and Referencing Environment

Deadline: April 6, 2014 (Sunday) 23:59

1 Introduction

The purposes of this assignment are to offer you a first experience of Ruby's Proc and a feature-rich programming language—Perl. The main focuses are referencing environment and dynamic scoping.

The assignment consists of three parts. First, you need to implement the classes for an Encryption/Decryption system using Ruby's Proc. Second, you need to implement the "Connect 4"game with Perl in object-oriented style, and experience the slight difference of object-oriented programming in Ruby and Perl. Third, three Perl scoping implementations of a Library Administration System is given. You need to understand the scoping in Perl and re-implement them in Ruby and C, and a Perl implementation with correct variable scoping. In the process, you will learn the difficulty of understanding codes written with dynamic scoping.

2 Task 1: Secure Communication System

In this task, you need to implement a simple secure communication system using Ruby's Proc. You have to ***follow strictly*** our rules to write the program.

2.1 Introduction

By secure communication, we mean that the communicated content between two entities could not be listened in by any unauthorized party. To achieve security, one way is to enforce encryption on the communicated content, referred to as *plainText*. In an encryption scheme, the *plainText* is encrypted using an algorithm, turning *plainText* into an unreadable *cipherText*. This is usually done with the use of an encryption key, which specifies how the *plainText* is to be encrypted. Any unauthorized party that can see the *cipherText* should not be able to recover the original message. An authorized party, however, is able to decode the *cipherText* using a decryption algorithm and a secret decryption key, to which unauthorized parties do not have access.

2.2 System design

In this task, you should use a symmetric-key encryption/decryption scheme to implement the secure communication system. In symmetric-key schemes, the encryption and decryption keys are the same.

A secure communication process is shown in Figure 1. Jimmy wants to have a secure conversation with Olivia regarding the assignment and final examination of CSCI 3180. After setting up the keys, encryption kernel, decryption kernel and password, Jimmy uses his *encrypt* method to encrypt the *plainText* into *cipherText*. And then Jimmy sends the *cipherText* together with his decryption kernel as executable codes together with the key in the referencing environment to Olivia. As soon as Olivia receives the *cipherText* and decryption kernel, she can use the *decrypt* method to get the original *plainText*. Olivia can send encrypted messages to Jimmy in a similar fashion. You can follow the arrows in Figure 1 to understand the communication process. Each oval stands for a method in the Jimmy and Olivia objects. The methods are invoked from top to bottom. Solid arrows represent data flow when Jimmy sends a message to Olivia, while broken

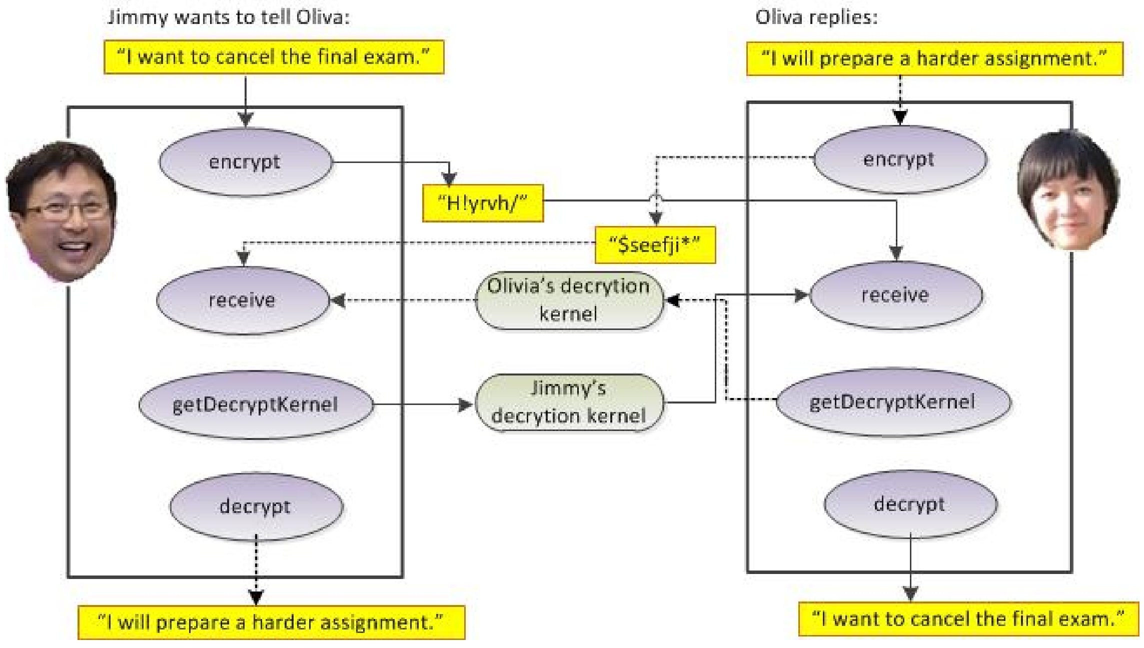
Jimmy wants to tell Oliva:
"I want to cancel the final exam."
encrypt
"H!yrvh/"
"$seefji*"
receive
Olivia's decrytion kernel
getDecryptKernel
Jimmy's decrytion kernel
decrypt
"I will prepare a harder assignment."
Oliva replies:
"I will prepare a harder assignment."
encrypt
receive
getDecryptKernel
decrypt
"I want to cancel the final exam."

Figure 1: Communication Process

arrows represent the reverse. An advantage of passing the decryption kernel as executable codes is that we do not have to send the key explicitly over the communication medium.

Each person uses two keys, a *key* and a *reserveKey*. The *key* is used for encryption and decryption. However, if the *key* has already been used successfully to decrypt a message once, the value of the *key* should be erased for safety concerns. To achieve that, we adopt a *reserveKey* to replace the *key* after it has been used once.

The password is used to protect one's own decryption kernel. To get other's decryption kernel, a user should provide the correct password.

Each person has an encryption kernel to encrypt the *plainText*. To distinguish the decryption kernel defined by oneself and the one received from others, you should maintain two decryption kernels. In this task, you should define *myDecryptKernel* to record one's own decryption kernel and *otherDecryptKernel* for the decryption kernel received from others. The encryption and decryption kernels contain the encryption and decryption algorithms respectively. The kernels should be implemented by Ruby's Proc, so that they could be passed as arguments to other methods. The *myDecryptKernel* should be returned from the *getDecryptKernel* method if provided with the correct password.

To receive a message and a decryption kernel, Olivia actually just uses a method *receive* to set the *cipherText* and *otherDecryptKernel* by the ones passed as parameters to this method.

2.3 Ruby Classes and Methods

Please follow the classes defined below in your Ruby implementation. To get a better understanding of this section, you can refer to Figure 1. You are free to add other variables, methods or classes.

1. **Class `Person`**
   In this class, an entity who wants to communicate securely with another is defined. You have to implement it with the following components:

   - **Instance Variable(s)**

`@key`
  - This is the key to encrypt and decrypt texts.

`@reserveKey`
  - This is a reserved key to replace `@key` if the `@key` has been used once.

`@password`
  - This is the password to check the validity of a request to get `@myDecryptKernel`.

`@otherDecryptKernel`
  - This is a Proc object received from others containing the decryption algorithm.

`@cipherText`
  - This is the received encrypted text.

- **Instance Method(s)**

  `initialize`
  - This method is already given:

```
def initialize(password)
  @password = password
  @key = 0
  @reserveKey = 0
  @otherDecryptKernel = -1
  @cipherText = ""
end
```

  `setKeys`
  - parameter(s): `key`, `reserveKey`.
  - Before encryption, a user needs to call this method first to generate and set values to `@key` and `@reserveKey`.

  `receive`
  - parameter(s): `cipherText`, `decryptKernel`.
  - This method should replace `@cipherText` by parameter `cipherText` and replace `@otherDecryptKernel` with `decryptKernel`.

  `encrypt`
  - parameter(s): `plainText`.
  - First, this method should check whether `@encryptKernel` is a Proc object. If not, the method should return the error message "`@encryptKernel is not a Proc object!`". Then it should check whether `plainText` is a string. If not, the method should return the error message "`plainText is not a string!`". If both checks are okay, then `@encryptKernel` should be activated.

  `decrypt`
  - parameter(s): None.
  - First, this method should check whether `@otherDecryptKernel` is a Proc object. If not, the method should return the error message "`@otherDecryptKernel is not a Proc object!`". Then it should check whether `@cipherText` is a string. If not, the method should return the error message "`@cipherText is not a string!`". If both checks are okay, `@otherDecryptKernel` should be activated in this method. **No parameter should be passed to this method**.

  `getDecryptKernel`
  - parameter(s): None.
  - This method should first ask users to enter a password and then check whether the entered one matches `@password` or not. If not, the method should print the error message "`Permission Denied!`" and return `-1`. If matched, the method should return `@myDecryptKernel`.

2. **Class** `Jimmy`
   This is a subclass of class Person in which `@encryptKernel` and `@decryptKernel` are initialized with specific encryption and decryption algorithms.

- **Instance Method(s)**

  `initialize`

  - Two additional instance variables are initialized here, i.e., `@encryptKernel` and `@myDecryptKernel`. Both of them are Proc objects, parameter `plainText` and `cipherText` should be passed to these two Proc objects respectively. The encryption algorithm is already given below as a method. You should turn it into a Proc Object and come up with the decryption one. Notice that the key should not be passed as a parameter for both `@encryptKernel` and `@myDecryptKernel`. Remember to use `super`.

    ```
    def encryptionAlgorithm(plainText)
        cipherText = ""
        plainText.each_char { | ch |
          newChar = (ch.ord ^ @key).chr
          cipherText += newChar
        }
        cipherText
    end
    ```

3. **Class** `Olivia`

   The components in this class are similar to those in class `Jimmy`.

- **Instance Method(s)**

  `initialize`

  - Two additional instance variables are initialized here, i.e., `@encryptKernel` and `@myDecryptKernel`. Both of them are Proc objects, parameter `plainText` and `cipherText` should be passed to these two Proc objects separately. Olivia's encryption algorithm is already given below as a method. You should turn it into a Proc Object and come up with the decryption one. Notice that the key should not be passed as a parameter for both `@encryptKernel` and `@myDecryptKernel`. Remember to use `super`.

    ```
    def encryptionAlgorithm(plainText)
        cipherText = ""
        plainText.each_char { | ch |
          newChar = (ch.ord + @key).chr
          cipherText += newChar
        }
        cipherText
    end
    ```

2.4 Execution Context

You should be able to run your program using the statements listed below:

```
jimmy = Jimmy.new("jimmy")
olivia = Olivia.new("olivia")
plainText = "I want to cancel the final exam."
jimmy.setKeys(2, 3)
cipherText = jimmy.encrypt(plainText)
puts "Olivia is receiving cipherText and the decryption kernel from Jimmy ..."
olivia.receive(cipherText, jimmy.getDecryptKernel)
puts "Olivia is decrypting the cipherText ..."
puts "For the first time, Olivia gets this result: #{olivia.decrypt}"
puts "For the second time, Olivia gets this result: #{olivia.decrypt}"
```

```
puts
plainText = "OK, I will prepare a harder assignment."
olivia.setKeys(4, 5)
cipherText = olivia.encrypt(plainText)
puts "Jimmy is receiving cipherText and decryption kernel from Olivia ..."
jimmy.receive(cipherText, olivia.getDecryptKernel)
puts "Jimmy is decrypting the cipherText ..."
puts "For the first time, Jimmy gets this result: #{jimmy.decrypt}"
puts "For the second time, Jimmy gets this result: #{jimmy.decrypt}"
```

2.5 Input and Output

The following gives the output using the execution context in the last section. We assume that the passwords for the `Jimmy` and `Olivia` objects are “jimmy” and “olivia” respectively.

```
Olivia is receiving cipherText and decryption kernel from Jimmy ...
Please enter the password of Jimmy:
jimmy
Olivia is decrypting the cipherText ...
For the first time, Olivia gets this result: I want to cancel the final exam.
For the second time, Olivia gets this result: H!v`ou!un!b`obdm!uid!gho`m!dy`l/

Jimmy is receiving cipherText and decryption kernel from Olivia ...
Please enter the password of Olivia:
olivia
Jimmy is decrypting the cipherText ...
For the first time, Jimmy gets this result: OK, I will prepare a harder assignment.
For the second time, Jimmy gets this result: NJ+Hvhkkoqdo`qd`g`qcdq`rrhfmldms-
```

3 Task 2: Connect 4

This is a small programming exercise for you to get familiar with Perl, which is a flexible and powerful programming language. In this task, you have to ***follow strictly*** the specifications about “Connect 4” given in Assignment 2.

4 Task 3: Library Administrative System

This task is to rewrite three Perl scoping implementations of a Library Administrative System in C and Ruby. You need to understand the Perl programs’ respective behavior and re-implement them in C and Ruby to produce the same behaviour. Among the three Perl implementations, only one of them gives the *correct* behavior. However, the variable declarations in that Perl program are not proper enough. Please also write a Perl program with proper variable declarations.

Three Perl programs of the system will be given to you. After finishing this task, you will have experienced a common but important programming language feature called scoping. And you will see the differences among C, Ruby and Perl, the last of which supports also dynamic scoping.

4.1 Background

As we described in the last assignment, Prof. Jimmy Lee and his group are building an advanced information system to take care of various administrative needs for CUHK. During the implementation of the Chinese University Administrative System, Jimmy, Olivia, Weixin were asked to implement the Library Administrative Sub-system unit in Perl. However, they came up with 3 similar implementations with exactly the same logic and algorithm but with distinct variable scoping declarations. Jimmy uses all "global"variables, Olivia uses all "my"variables, and Weixin uses all "local"variables. Even worse, the programming language for the project will be changed to either C or Ruby all of a sudden, so that all previous work need to be re-implemented. You are Jimmy's excellent students who are willing to help Jimmy share his stress.

4.2 Task Requirement

The requirement is simple, you need to re-implement the three Perl programs in C and Ruby to produce identical behavior. You are *not* allowed to touch the program logic, and can only fiddle with the variable scoping and type declarations and initializations. In addition, all of Jimmy's, Olivia's and Weixin's programs are "wrong"in variable scoping declaration. You need to give a Perl program with proper declarations with correct behavior. Just as before, you will also be evaluated on your programming style.

5 Report

Your simple report should answer the following questions within THREE A4 pages:

1. Using your codes for Task 1, discuss the advantages of Ruby Proc, and explain the execution of your program in terms of scoping and referencing environment.
2. Using codes for Task 2, discuss features of object-oriented Perl that are different from the object-oriented features of Ruby.
3. Using codes for Task 3, compare the differences of scoping in C, Perl and Ruby. Also, explain and justify the variable type declarations of your *correct* Perl programs.
4. Using codes for Task 3, explain whether dynamic scoping is needed in a programming language.

6 Submission Guidelines

Please read the guidelines CAREFULLY. If you fail to meet the deadline because of submission problem on your side, marks will still be deducted. So please start your work early!

1. In the following, **SUPPOSE**

   your name is *Chan Tai Man*,
   your student ID is *1155234567*,
   your username is *tmchan*, and
   your email address is *tmchan@cse.cuhk.edu.hk*.

2. In your source files, insert the following header. REMEMBER to insert the header according to the comment syntaxes of C, Ruby and Perl.

```
/*
 * CSCI3180 Principles of Programming Languages
 *
 * --- Declaration ---
 *
 * I declare that the assignment here submitted is original except for source
 * material explicitly acknowledged.  I also acknowledge that I am aware of
 * University policy and regulations on honesty in academic work, and of the
 * disciplinary guidelines and procedures applicable to breaches of such policy
 * and regulations, as contained in the website
 * http://www.cuhk.edu.hk/policy/academichonesty/
 *
 * Assignment 3
 * Name : Chan Tai Man
 * Student ID : 1155234567
 * Email Addr : tmchan@cse.cuhk.edu.hk
 */
```

The sample file header is available at

`http://www.cse.cuhk.edu.hk/~csci3180/resource/header.txt`

3. Make sure you compile and run the C programs without any problem with g++ on the department's Linux machines.

4. Make sure you run the Ruby programs without any problem on the department's Linux machines.

5. Make sure you run the Perl programs without any problem on the department's Linux machines.

6. The report should be submitted to VeriGuide, which will generate a submission receipt. The report and receipt should be submitted together with your C, Ruby and Perl codes in the same ZIP archive.

7. For task 1, you should name your program as `rubyProc.rb`.

8. For task 2, your programs should be named as `Player.pm`, `Computer.pm`, `Human.pm`, `connect4.pl`.

9. For task 3, you should submit seven files: three Ruby, three C and one Perl programs. The Ruby and C program files should follow the same names as the corresponding Perl program files, by replacing only the suffices with `.rb` and `.c` respectively. The Perl program file should be named "`correct.pl`".

10. The filename of the report should be `report.pdf`. The name of the VeriGuide receipt should be `receipt.pdf`.

11. All file naming should be followed strictly and without quotes.

12. `Tar` your source files to `username.tar` by

    ```
    tar cvf tmchan.tar rubyProc.rb Player.pm Computer.pm Human.pm\
    connect4.pl my.rb local.rb global.rb my.c local.c global.c\
    correct.pl report.pdf receipt.pdf
    ```

13. `Gzip` the `tarred` file to `username.tar.gz` by

    ```
    gzip tmchan.tar
    ```

14. `Uuencode` the `gzipped` file and send it to the course account with the email title "HW3 *studentID yourName*" by

```
uuencode tmchan.tar.gz tmchan.tar.gz \
| mailx -s "HW3 1155234567 Chan Tai Man" csci3180@cse.cuhk.edu.hk
```

15. Please submit your assignment using your Unix accounts.

16. An acknowledgment email will be sent to you if your assignment is received. **DO NOT** delete or modify the acknowledgment email. You should contact your TAs for help if you do not receive the acknowledgment email within 5 minutes after your submission. **DO NOT** re-submit just because you do not receive the acknowledgment email.

17. You can check your submission status at

    `http://www.cse.cuhk.edu.hk/~csci3180/submit/hw3.html`.

18. You can re-submit your assignment, but we will only grade the latest submission.

19. Enjoy your work :>