Université d'Ottawa
Faculté de génie

School of Electrical Engineering and Computer Science

uOttawa

L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

École de science informatique et de génie électrique

CSI2120 Programming Paradigms

FINAL Evaluation

Length of Examination: 3 hrs **April 16, 2020, 14:00-17:00**

Professor: Jochen Lang

You must abide by and have acknowledged the declaration of integrity.

You must upload your solutions as three source code files with **comments** to Brightspace. If you experience any technical difficulties, you must notify the Professor immediately.

Question	Marks	Out of
1		13
2		12
3		13
Total		38

Question 1 Functional Programming in Scheme

Please note:

- *You are not allowed to use set! in answering this question.*
- *You must comment your code.*

Given is the following list of names:

```
(define names
     '("marie" "jean" "claude" "emma" "sam" "tom" "eve" "bob"))
```

You are asked to produce a function that will allow you to randomly pick N names from this list.

For example:

```
> (winner names 3)
("claude" "jean" "eve")
> (winner names 3)
("tom" "sam" "jean")
> (winner names 2)
("eve" "jean")
```

The function winner uses the built-in random function to generate random integers.

```
> (random 8) ; generates an integer between 0 and 8 exclusively
5
> (random 8)
4
> (random 8)
6
```

To achieve the winner function you must proceed in three steps. Each of these steps are independent so even if you don't have the answer to one step you can proceed with the subsequent steps.

a) Write a helper function `first` in order to obtain the first N people from a list. The function is to return a list of the N first elements of the input list.

```
> (first 3
         '("tom" "jean" "eve" "claude" "sam" "marie" "emma" "bob"))
("tom" "jean" "eve")
> (first 5
         '("tom" "jean" "eve" "claude" "sam" "marie" "emma" "bob"))
("tom" "jean" "eve" "claude" "sam")
```

b) Create a function `insertAt` that inserts an element in a list at a given position

```
> (insertAt 'a '(b d e g) 2)
(b d a e g)
> (insertAt 'a '(b d e g) 1)
(b a d e g)
> (insertAt 'a '(b d e g) 20)
(b d e g a)
> (insertAt 'a '(b d e g) 0)
(a b d e g)
```

c) For the draw of the winner, the original list is shuffled by using the `insertAt` function. That is, by inserting the first element of the list in the rest of the list at a random position, i.e.,

```
> (insertAt (car names) ; insert the first element
          (cdr names) ; in the rest of the list
            (random (- (length names) 1))) ; at a random position
 ("jean" "claude" "emma" "sam" "marie" "tom" "eve" "bob")
```

We repeatedly apply the above to the new list returned. By repeating several times, we have the elements in a completely random order.

Complete the `shuffle` function which calls the `insertAt` function recursively n times

```
(define (shuffle lst n)
   (if (= n 0) lst
       ( …

> (shuffle names 20)
("tom" "jean" "eve" "claude" "sam" "marie" "emma" "bob")
```

Finally, the `winner` function is:

```
(define (winner lst n)
    (first n (shuffle lst 20)))
> (winner names 3)
("sam" "tom" "emma")
```

<u>Question 2</u> Logic Programming in Prolog

Please note:

- *You must comment your code.*

The following are the daily weekend temperatures for March 2020
`[-4, 1, 6, 4,-2,-4, 0, 7, 8]`

The normal temperatures based on historical averages for these dates are:

`[-1, 0, 0, 2, 2, 4, 4, 6 ,6]`

A database containing this information is given below.
(We only need one month but you can assume that this information exists for other months too):

```
; weekends(Month, Year, WeekEndTemperature, Normals)
weekends(march,2020, [-4, 1, 6, 4,-2,-4, 0, 7, 8],
                     [-1, 0, 0, 2, 2, 4, 4, 6 ,6]).
```

a) Design a predicate `difference/3` calculating the difference between the temperature of weekends in March 2020 and the normal temperature for these days.

```
?- difference([-4, 1, 6, 4,-2,-4, 0, 7, 8],
                      [-1, 0, 0, 2, 2, 4, 4, 6 ,6], D)
D= [-3, 1, 6, 2, -4, -8, -4, 1, 2]
```

b) Design a predicate `positive/2` giving the number of positive numbers in a list.

```
?- positive ([-3, -1, 6, 2, -4, -8, -4, 1, 2], N).
N= 4.
```

c) Use the `difference/3`, `positive/2`, and `weekends/4` predicates to design a `niceMonth/2` predicate returning true if at least half of the weekend days for a month were above normal.

Note that the number of weekend days in a month is variable. You may find the built-in predicate `length/2` useful.

```
? - length ([a, b, c], L). L = 3.
```

```
?- niceMonth(march,2020).
 true.
```

Question 3 Imperative Programming in Go

The following program is to simulate a simple real estate market. You can run the attached program and you will see a simulation of a bidding process with 7 `Buyers` and 3 `Sellers` with one `Listing` each. Note that bidding is randomized and you will see different results every time.

The program defines 5 main types:

- Seller representing the ones who have their property for sale.
- Buyer representing the ones who wants to buy a property
- Condo, House, and TownHouse representing the properties to be sold

Note: you do not need to look at the code of the `Seller` and `Buyer` types in order to answer to the below questions.

The main function of this program simulates the bidding process in which Buyers bid on properties sold by Sellers. This process is subdivided into steps as follows

Note: Each of these steps are independent so even if you don't have the answer to one step you can proceed with the subsequent steps.

Answer the questions a) to h) below

STEP 1: Listings of properties to be sold.

The current process has a listing of 3 condos, consider the `listings` slice in the main routine.

```
listings := []*Condo{{"Goulburn Ave 1120", 750000, false, 900},
                     {"Summerset Street 10", 950000, false, 850},
                       {"Wilbord Avenue 999", 1250000, false, 1250}}
```

In general, listings should also contain House and TownHouse instances.

a) Embedding: Create the structure `ListingInfo` to contain all fields common for `Condo`, `House` and `TownHouse`, remove these common fields and instead embed `ListingInfo` in `Condo`, `House` and `TownHouse`.

```
// ListingInfo structure
type ListingInfo struct {
   // code to be added here
   // see line 15 in Go code provided
}
```

b) **Interfaces:** Change the declaration of the listings variable to be a slice of `RealEstate` with `RealEstate` being an interface that the Condo, House and TownHouse types must implements

```
listings := []RealEstate{
              {"Goulburn Ave 1120", 750000, false, 900},
               {"Summerset Street 10", 950000, false, 850},
                {"Wilbord Avenue 999", 1250000, false, 1250}}
```

c) Add one `House` and one `TownHouse` to `listings` (so there will be now 5 properties listed)
- TownHouse of 2 levels at "Elgin 123" at $2100000
- House at "Maplewood 889" at $850000 with lot size of 50 x 110

STEP 2: Sellers for every listing (line 185)

```
sellers := []*Seller{NewSeller("Eve", listings[0]),
                     NewSeller("Monica", listings[1]),
                     NewSeller("Ramon", listings[2])}
```

d) Add a seller for the newly added `House` and `TownHouse` of question c).to `listings` (so there will be now 5 sellers listed).
- Paul the TownHouse seller
- Mary the House seller

Note: If you were not able to complete Step 1 simply add two new Condo in `listings` and their two sellers here.

STEP 3: Buyers (line 190)

You don't need to change anything here.

```
buyers := []*Buyer{NewBuyer("Zara"), NewBuyer("Jim"),
                   NewBuyer("Claude"), NewBuyer("Emilie"),
                   NewBuyer("Amelie"), NewBuyer("Ali")}
```

STEP 4: Bidding process (line 194)

The current bidding process will need to be concurrent but currently it is completely sequential.

Note: You can answer this part with or without the `ListingInfo` structure or the `RealEstate` interface.

e) Go Routines:

In the for loop below of the bidding process (line 194), each buyer tries to buy a property one after the other (sequentially). Introduce a go routine (possible a lambda function) such that each buyer's bidding process in the main routine runs concurrently.

```
for _, buy := range buyers { // for each buyer
    // Buyers need to bid concurrently
   // introduce a Go rountine here
```

f) **Channels:**
The second for loop (line 200) is for a buyer to propose increasing prices for each property in order to buy one:

```
// loop by generating different amount
for amount, valid := buy.nextBid(); valid;
                          amount, valid = buy.nextBid() {
```

This loop stops if one `Seller` accepts the amount offered or if the Buyer does not want to bid a higher amount.

Note: You do not need to understand how prices are generated.

In the current iterative version, a buyer checks if a seller accepts the offer by calling a function (line 207):

```
// Is bid accepted? – if true is returned then close deal
if s.acceptBid(amount) {
```

But in the `Seller` type there is a `OfferChan` channel defined to receive the amount offered and a `ResponseChan` boolean channel to send the response. Each time a seller instance is created, the following go function is also called (line 142):

```
go func() {
     for {
          select {
          case offer := <-s.OfferChan:
               s.ResponseChan <- s.acceptBid(offer)
          }
     }
}()
```

Replace the call to the `acceptBid` function by instead communicating with a Seller through these channels for effective synchronization.

STEP 5: Synchronization (line 220)

You don't need to change anything here.

The program will only exit once the bidding process is complete, i.e., there are no more active buyers or all objects are sold.