CSc 600-01 (Section 1)

# Homework 4 - Functional Programming

*prepared by Ilya Kopyl*

# CSc 600 Homework 4 - Scheme and Functional Programming

Ilya Kopyl

April 24, 2018

*Homework is prepared in LaTeX with TeXShop editor (under GNU GPL).*

**1. The concept of first class objects is fundamental for Scheme programming. In particular, in Scheme language any function is a first class object. The main properties of a function as a first class object are exemplified by answering the following questions:**

a) The first class object may be expressed as an anonymous literal value (constant). Show an example of the anonymous function and its use.

b) The first class object may be stored in variables (i.e. it may have a symbolic name). Show examples of defining and using named functions.

c) The first class object may be stored in data structures. Show an example of a data structure (e.g. a list) that contains functions.

d) The first class object may be comparable to other objects for equality. Show an example of comparing functions and lists for equality.

e) The first class object may be passed as parameter to procedures/functions. Show an example of passing function as an argument to another function.

f) The first class object may be returned as result from procedures/functions. Show an example of returning a function as a result of another function.

g) The first class object may be readable and printable. Show examples of:

- reading function(s) from keyboard,
- reading function(s) from a file,
- displaying a function.

The answer is listed on the pages TBD through TBD.

Problem 1a: show an example of the anonymous function and its use:

At the very basic case, an anonymous function is nothing more but a lambda expression - a body of a function. Since it is not associated with any identifier (i.e. when it is unnamed), we may use it only when we explicitly write it inside of other expressions:

```
> ((lambda(x) (* x x)) 2)
4
```

Also, an anonymous function can be used in cases when we otherwise would have to define an inner function (inside another function):

```
> (define (add-num-to-each-element-in-list num lst)
    (map (lambda (x) (+ x num)) lst))

> (add-num-to-each-element-in-list 10 '(1 2 3))
(11 12 13)
```

Problem 1b: show examples of defining and using named functions:

Since we already know what a lambda function is, we can now take it and associate it with a name (identifier) - and then we could use just this name to evaluate any expression with it:

```
> (define square (lambda(x) (* x x)))

> square
#<procedure:square>

> (square 2)
4
> (square 4)
16
```

- thus the function definition is a process of binding a lambda expression to some identifier. For simplicity and convenience we could make the same function definition with the use of syntactic sugar and omit 'lambda' keyword. Semantically, such function definition would still remain the same:

```
> (define (square x) (* x x))

> (square 2)
4
> (square 4)
16
```

We could also take our defined function square and store it in a variable:

```
> (define a sqr)

> a
#<procedure:sqr>

> (a 2)
4
> (a 4)
16
```

The reason why we define functions is to follow the Single Responsibility Principle, when each function is doing only one thing. But at certain point we would need to do a function composition, i.e. to define a function that is composed of series of expressions that use previously defined functions:

```
> (define (sum-of-squares lst)
    (apply + (map square lst)))

> (sum-of-squares '(1 2 3 4))
30
```

Problem 1c: show an example of a data structure (e.g. a list) that contains functions:

If we think of a function as an entity that is responsible for only one operation/modification, then chaining different functions to each other to perform sequential modification of the same data can be viewed as a sort of Henry Ford's conveyor belt. In the previous code we considered a case when the number, kinds of functions and their order of evaluation are known and determined, but a data is not. But let's consider a situation when we initially don't know a number, nor kinds of functions, nor their order of evaluation at all. We would need to figure out how to modify a given data dynamically, at runtime. The approach is to pass a list of functions as an extra argument to our master function. With the help of a tail recursion we could take one function at a time from that list, and evaluate it with our data, provided that the arity of these functions match with the number of data arguments. But I digressed. There are two data structures to store functions in: a list and a quoted list. Let's consider both approaches:

```
; functions we are going to work on
> (define (square x)
    (* x x))

> (define (double x)
    (* x 2))
```

```
; example of a quoted list: functions turned into symbols
> (quote (square double))
(square double)
> '(square double)        ; syntactic sugar equivalent
(square double)

> (symbol? (car '(square)))
#t

; example of a list of functions:
> (list square double)
(#<procedure:square> #<procedure:double>)

> (procedure? (car (list square)))
#t


; we can assign both lists to variables:
> (define lst-of-fun1 '(square double))

> (define lst-of-fun2 (list square double))


; differences in use:
> ((car (list square)) 5)
25

> ((eval (car '(square))) 5)
25
```

Problem 1d: show an example of comparing functions and lists for equality:

To answer this question we must first understand what equality means when it comes to comparing functions or lists with each other. For any object to be equal to another object in Scheme it either needs to point to the same location in memory as the other one, or to have the same type and value, or to have the same, equal numerical value.

Per Revised Report on the Algorithmic Language Scheme (r7rs):

- The eqv? procedure defines a useful equivalence relation on objects. Briefly, it returns #t if obj1 and obj2 are pairs, vectors, bytevectors, records, or strings that denote the same location in the store; or if obj1 and obj2 are procedures whose location tags are equal.

- The equal? procedure, when applied to pairs, vectors, strings and bytevectors, recursively compares them, returning #t when the unfoldings of its arguments into (possibly infinite) trees are equal (in the sense of equal?) as ordered trees, and #f otherwise. It returns the same as eqv? when applied to booleans, symbols, numbers, characters, ports, procedures, and the empty list. If two objects are eqv?, they must be equal? as well. In all other cases, equal? may return either #t or #f.
- The eq? predicate is similar to eqv? except that in some cases it is capable of discerning distinctions finer than those detectable by eqv?. On symbols, booleans, the empty list, pairs, and records, and also on non-empty strings, vectors, and bytevectors, eq? and eqv? are guaranteed to have the same behavior. On procedures, eq? must return true if the arguments? location tags are equal. On numbers and characters, eq?'s behavior is implementation-dependent, but it will always return either true or false. On empty strings, empty vectors, and empty bytevectors, eq? may also behave differently from eqv?.

By mathematical definition of a function, the only valid way to compare two functions for their equivalence is to verify that for every valid input both functions produce the same output, and that both functions also have the same domain and range. This means that a function equality can only be defined correctly in terms of operational equivalency; that is, the implementation doesn't matter, only the behavior does. This, of course, is an undecidable problem in any nontrivial language. It is impossible to determine if any two functions are operationally equivalent because, if we could, we could solve the halting problem.

Let's consider a series of experiments to prove this point:

```
> (define (square x)
    (* x x))
> (eq? square square)
#t
> (equal? square square)
#t
> (eqv? square square)
#t


> (define a square)
> (eq? a square)
#t
> (equal? a square)
#t
> (eqv? a square)
#t
```

```
> (define (sqr x)
    (* x x))
> (eq? square sqr)
#f
> (equal? square sqr)
#f
> (eqv? square sqr)
#f
```

- as you can see, if two identifiers both point to the same object (a function) in memory, the result of comparison of these two identifiers with any of the predicates eq?, equal?, eqv? would result in true. However, there are no means to check the equivalence of functions located in two different locations in memory, even if they consist of the same sequence of expressions.

Examples of comparing lists for equality:

```
; base case: an empty list
> (equal? '() '())
#t
> (eq? '() '())
#t
> (eqv? '() '())
#t

; a list of 1 symbol
> (equal? '(a) '(a))
#t
> (eq? '(a) '(a))
#f
> (eqv? '(a) '(a))
#f

; same as:
> (equal? (list 'a) (list 'a))
#t
> (eq? (list 'a) (list 'a))
#f
> (eqv? (list 'a) (list 'a))
#f


> (equal? '(a (b) c) '(a (b) c))
#t
> (eq? '(a (b) c) '(a (b) c))
#f
> (eqv? '(a (b) c) '(a (b) c))
#f
```

```
> (define list1 '(1 2 3))
> (define list2 '(1 2 3))

> (equal? list1 list2)
#t
> (eq? list1 list2)
#f
> (eqv? list1 list2)
#f

> (define list11 list1)

> (equal? list11 list1)
#t
> (eq? list11 list1)
#t
> (eqv? list11 list1)
#t
```

The code listing of main program:

```
#include <stdio.h>
#include <assert.h>
#include "functions.h"

unsigned int maxlen(int *, unsigned int);
```

Auxiliary functions (in separate file "functions.c"):

```
#include <stdio.h>
#include "functions.h"
```

The result of the program execution:

```
Array a:    1  1  1  2  3  3  5  6  6  6  6  7  9
```

**2. Integer plot function (find a smart way to code big integers)**

Write a program BigInt(n) that displays an arbitrary positive integer n using big characters of size 7x7, as in the following example for BigInt(170):

```
  @@      @@@@@@@@   @@@@@
 @@@            @@  @@    @@
```

Write a demo main program that illustrates the work of BigInt(n) and prints the following sequence of big numbers 1, 12, 123, 1234,..., 1234567890, one below the other.

The answer is listed on the pages 7 through 9.

The code listing of the two-dimensional array that stores bit pattern of each BigInt digit. It is declared in the global space (outside of any function).

```
#define NUMBER_OF_ROWS 8

/**
 * Digits are stored as bit patterns of 8-bit unsigned integer (char) numbers.
 *
```

```
 * Each digit requires just 8 bytes of storage - which is polynomially smaller
 * than the storage in brute-force approach where each digit is represented by
 * a 2D array of 8x8 characters, with 64 bytes of storage per digit.
 */
```

Main program, excluding the declaration of BIG_DIGITS array:

```
#include <stdio.h>
#include <math.h>

void BigInt(unsigned int);
unsigned int getNumberOfDigits(unsigned int);

#define NUMBER_OF_BITS 8
#define NUMBER_OF_ROWS 8
```

The result of the program execution:

```
       Blah-blah-blah
asdasd 12 123 12 3123

123123
```

### 3. Array processing (elimination of three largest values) (one of many array reduction problems)

The array a(1..n) contains arbitrary integers. Write a function reduce(a, n) that reduces the array a(1..n) by eliminating from it all values that are equal to three largest different integers. For example, if a=(9, 1, 1, 6, 7, 1, 2, 3, 3, 5, 6, 6, 6, 6, 7, 9) then three largest different integers are 6, 7, 9, and after reduction the reduced array would be a=(1, 1, 1, 2, 3, 3, 5), n=7. The time complexity of the solution should be in O(n).

The answer is listed on the pages 11 through 13.

The code listing of the entire program for problem #3:

```c
#include <stdio.h>
#include "functions.h"

unsigned int reduce(int *, unsigned int);
void findTop3MaxValuesInArray(int *, unsigned int, int *, int *, int *);
void nullifyTop3MaxValuesInArray(int *, unsigned int, int, int, int);
unsigned int moveZeroesToEndOfArray(int *, unsigned int);
```

Auxiliary functions (in separate file "functions.c"):

```c
#include <stdio.h>
#include "functions.h"
```

The result of the program execution:

```
$ gcc -Wall -std=c99 hw2-problem3.c functions.c -O3
$ ./a.out
```

**4. Iteration versus recursion (an opportunity for performance measurement)**

Make a sorted integer array a[i]=i, i=0,...,n-1. Let bs(a, n, x) be a binary search program that returns the index i of the array a[0..n-1] where a[i]=x. Obviously, the result is bs(a, n, x)=x, and the binary search function can be tested using the loop

```
for (j=0; j < K; j++)
    for (i=0; i < n; i++)
        if (bs(a, n, i) != i)
            cout << "\nERROR";
```

Select the largest n your software can support and then K so that this loop with an iterative version of bs runs 3 seconds or more. Then measure and compare this run time and the run time of the loop that uses a recursive version of bs. Compare these run times using maximum compiler optimization (release version) and the slowest version (minimum optimization or the debug version). If you use a laptop, make measurements using AC power, and then the same measurements using only the battery. What conclusions can you derive from these experiments? Who is faster? Why?

The answer is listed on the pages 15 through 19.

The code listing of the entire program for problem #4:

```c
#include <stdio.h>
#include <time.h>

#define K 1000                          // system-dependent constant

void initializeArray(int *, int);
int ibs(int *, int, int);
int rbs(int *, int, int, int);
double ibsTest(int *, int);
double rbsTest(int *, int);
```

The result of the program execution with different setup & compiler optimizations:

```
# connected to charger, no Wi-Fi, no monitors connected:

$ gcc -Wall -std=c99 hw2-problem4.c -O0
$ ./a.out
Running time of iterative Binary Search: 12.815268 seconds.
```

Initially, with minimal compiler optimization, the implementation of recursive Binary Search

```
$ gcc -Wall -std=c99 hw2-problem4.c -O0
$ ./a.out
Running time of iterative Binary Search: 12.409296 seconds.
```

One possible explanation to that counterintuitive phenomena could be related to behavior defined in operating system. When it activates a power savings mode, it is very likely that certain background processes halt, thus making my program run faster by tens of milliseconds.

**5. Iteration versus recursion (another opportunity for performance measurement)**

Write a recursive function Frec(n) that computes Fibonacci numbers. Then write an iterative version of Fibonacci number function Fit(n). Functions Frec(n) and Fit(n) return the same value but with different performance.

Write the main program that discovers the value N10 so that Frec(N10) runs on your machine exactly 10 seconds. Then measure the run time of Fit(N10) and compute how many times is Fit(N10) faster than Frec(N10). Show what is N10 on your machine.

Notes:

1. When you measure the speed, your machine should be disconnected from the Internet, it should use the AC power supply, and it should run only one program (your performance measurement program).

2. In C++ you can measure current time in seconds using the following function:

```
double sec(void)
{
    return double(clock()) / double(CLOCKS_PER_SEC);
}
```

To measure the run time of fast programs you must repeat them many times inside a loop. Take care to eliminate the overhead generated by the loop.

The answer is listed on the pages 20 through 22.

The code listing of the entire program for problem #5:

```
#include <stdio.h>
#include <zconf.h>
#include <time.h>

int Frec(int);
int Fit(int);

double findN(int *);

// using the function pointer to avoid unnecessary code repetition:
double benchmarkFibFunction(int (*f)(int), int);
```

The result of the program execution with different compiler optimizations:

```
$ gcc -std=c99 -Wall hw2-problem5.c -O0
$ ./a.out && ./a.out
Depending on the compiler optimizations, the results may vary. Please wait...
```

As you can see, a slightly more aggressive optimization enables Frec function to find