Sabancı University
Faculty of Engineering and Natural Sciences

CS305 Programming Languages

Homework 4

Due: 20 May 2023, 11:55 PM.

1 Introduction

In this homework you will implement various Scheme procedures that will hopefully give you a better grasp on the language.

2 Some Helpful Hints

In this section, some useful hints that you can use for debugging purposes are given. However, any imperative features used for debugging purposes need to be removed before submitting the homework.

You will see that in different part of this homework we ask you to produce an error under some conditions. All these errors must be produced by using this error procedure of Scheme. We will catch the error messages generated by the error procedure in our automatic grading script. Therefore it is important that you use the error procedure for producing the error messages. All your error messages should be in the form:

ERROR305: {OPTIONAL ERROR MESSAGE}

This means that all your error messages must start with the keyword ERROR305 followed by a colon followed by an optional error message of your choice.

```
1 ]=> (define add (lambda (n1 n2)
           (if (and (number? n1) (number? n2))
               (+ n1 n2)
               (error "ERROR305: actuals are not numbers" ))))

1 ]=> (add 3 4)
;Value: 7

1 ]=> (add 'a 4)
;ERROR305: actuals are not numbers
;To continue, call RESTART with an option number:
; (RESTART 1) => Return to read-eval-print level 1.

2 error>
```

The error procedure will put you in the error prompt. For debugging purposes, you may want to print out the values of certain expressions. Scheme has the built in procedure display for such a purpose.

```
1 ]=> (display 5)
5
;Unspecified return value

1 ]=> (define x 3)
;Value: x

1 ]=>  (display x)
3
;Unspecified return value

1 ]=> (define x '(1 2 a))
;Value: x

1 ]=> (display x)
(1 2 a)
;Unspecified return value
```

Note that, since there is no sequential execution of statements in the sense that you are used to, the place that you'll put these display statements may not be apparent. To show an example of typical usage, let us assume that, for the add procedure given above, we want to see the values of the non-number actuals. In this case, add procedure can be declared in the following way:

```
1 ]=> (define add (lambda (n1 n2)
           (if (and (number? n1) (number? n2))
               (+ n1 n2)
               (let* (
                      (dummy1 (if (number? n1)
                                  (display "::number::")
                                  (display n1)))
                      (dummy2 (if (number? n2)
                                  (display "::number::")
                                 (display n2)))
                      )
                      (error "ERROR305: actuals are not numbers")))))
;Value add

1]=> (add 3 4)
;Value: 7

1 ]=> (add 3 'a )
::number::a
;ERROR305: actuals are not numbers
;To continue, call RESTART with an option number:
; (RESTART 1) => Return to read-eval-print level 1.
2 error>
```

Note that let could have been used instead of let*, however, in that case the order of bindings would not be deterministic, hence it would be more difficult to understand which actual parameter caused the problem.

In Scheme, you can comment out parts of the source by using semicolon (;) character. It will comment out the rest of the line. This can be used as a trick for your testing purposes in the following way. Assume that, you are trying to implement the add procedure given above. Rather than directly typing it in the Scheme interpreter, it is easier for the development purposes, to write the code in a separate text file, let's say add.scm and then load it into the interpreter. Furthermore, you can also add some test cases at the end of the file. When MIT Scheme interpreter loads a file, it will print the value of the last expression that appears in the input file. More explicitly, let us assume that you have a file named add.scm with the following content:

```
; Adds to arguments if they are both numbers.
; If a nonnumber argument is seen, it produces
; an error

(define add (lambda (n1 n2)
     (if (and (number? n1) (number? n2))
         (+ n1 n2)
         (let* (
               (dummy1 (if (number? n1)
                           (display "::number::")
                           (display n1)))
               (dummy2 (if (number? n2)
                           (display "::number::")
                           (display n2)))
               )
               (error "ERROR305: actuals are not numbers")))))


; tests for the procedure add
; (add 3 5)
; (add 4 5)
; (add 4 'a)
```

You can load this file into the interpreter as follows:

```
1]=> (load "add.scm")
;Loading "add.scm" -- done
;Value:  add
```

Note that, it prints out a value add, which is the identifier defined by the last expression in the file. Now assume that we have removed the semicolons in front of the `(add 3 5)` and `(add 4 5)` in the file `add.scm`. If we load the file again, here is what will happen:

```
1]=> (load "add.scm")
;Loading "add.scm" -- done
;Value:  9
```

As you see, the value 9 is printed which is the value of the last expression in the file (which is the result of the last expression `(add 4 5)`). Hence by using comments, you can test your implementation by activating / deactivating some test cases.

3 Triangles

In this homework, you will implement some procedures related to the following concepts on triangles.

- **Triple**: A list of three numbers such as (5 4 8). We will use this form of triples to represent triangles. You can think that these numbers as the lengths of the sides of a triangle. For example, the triple (5 4 8) indicates a triangle having sides of length 5, 4, and 8. A triple is *sorted* if the second element is not smaller than the first element, and the third element is not smaller than the second element. For example (5 4 8) is not a sorted triple, but (4 5 8) is sorted triple, so is (5 5 8).
- **Triangle rule**: The sum of the lengths of any two sides of a triangle must be strictly greater than the third side. For example the triple (15 4 8) does not correspond to a triangle, since we have $4 + 8 < 15$.
- **Pythagorean theorem**: The sum of the areas of the two squares on the legs (a and b) equals the area of the square on the hypotenuse (c). $a^2 + b^2 = c^2$.

4 Flow of the procedures

In terms of the main flow of the program, one input will be given to the main procedure. This input must be a list consisting of arbitrary number of triples and it should not be empty. It is important that you can not make any assumptions about the number of triples in the list. Also, you can not assume anything about the content of the elements in the list. Hence, there should be an input check.

Then, if the input is correct, you should sort the numbers in each of the triples in ascending order. After sorting the numbers, you should filter the triples that satisfy the "Triangle rule" first and then you should filter the triples that satisfy the "Pythagorean theorem" described in section 3.
In the last step, you should sort the triples (Pythagorean triangles) that you filtered in the previous step according to the areas of the triangles, again in ascending order.

5 Scheme procedures to be implemented

You will implement the following procedures in Scheme. We will represent the triangles as lists of Scheme numbers (triples). For example a triangle will ne represented as the Scheme list of numbers **(3 4 5)** with three elements where the first element is the number **3**, the second element is the number **4** and the third element is again the number **5**. In the Scheme procedures to be implemented below, whenever we talk about a "triple", please assume that it is such a list. In addition, in some of the Scheme procedures you will implement, you will use the list of triples defined in the

section 4. Whenever we talk about a "tripleList", please assume that it is such a list. The procedures to be implemented are really easy. You can define helper procedures to be used in your code. You can also use the procedure **main-procedure** defined below to understand the flow of the program and the error procedure introduced above to produce the errors. However, do not use any built-in procedure other than the ones we've covered in our lecture notes. Below is the list of procedures to be implemented:

```
;- Procedure: main-procedure
;- Input : Takes only one parameter named tripleList
;- Output : Returns list of triples according to the scenario
; described in section 3.
; Returns an error if the given parameter is not a tripleList.
(define main-procedure
    (lambda (tripleList)
        (if (or (null? tripleList) (not (list? tripleList)))
            (error "ERROR305: the input should be a list full of triples")
            (if (check-triple? tripleList)
                (sort-area (filter-pythagorean (filter-triangle
                (sort-all-triples tripleList))))
                (error "ERROR305: the input should be a list full of triples")
             )
        )
    )
)

;- Procedure: check-triple?
;- Input : Takes only one parameter named tripleList
;- Output : Returns true if the given parameter consists of triples,
; returns false if otherwise.
;- Hint: You can assume that the given input to this procedure is
; a list and it is not null.
;- Examples :
; (check-triple? '((1 2) (3 4 5))) ------> evaluates to false since '(1 2)
 has only 2 elements
; (check-triple? '((5 12 12) (6 6 6) ())) ----> evaluates to false since
 '() is empty
; (check-triple? '((5 3 9) (9 55 32) ('a 28 67))) -----> evaluates to false
 since 'a is not a number
; (check-triple? '((5 12 13) (3 4 5) (16 63 65) (12 35 37))) --> evaluates to true
(define check-triple?
     (lambda (tripleList)
        ...
     )
)
```

```
;- Procedure: check-length?
;- Input : Takes two parameters as inTriple and count
;- Output : Returns true if the length of the given list
; equals to count, returns false if otherwise.
;- Hint: You can assume that the given input (inTriple) to this procedure is
; a list.
;- Examples :
; (check-length? '('a 'bc) 2) --> evaluates to true
; (check-length? '(3 4 5) 3) --> evaluates to true
; (check-length? '(2 3 4 5) 3) --> evaluates to false since the length
 of the list is not 3
; (check-length? '(3 4 5) 4) --> evaluates to false since the length of
 the list is not 4
(define check-length?
    (lambda (inTriple count)
        ...
     )
)

;- Procedure: check-sides?
;- Input : Takes only one parameter named inTriple
;- Output : It returns true if all of the elements in the given
; list are numbers and each of the numbers is greater than zero.
; It returns false if otherwise.
;- Hint: You can assume that the given input to this procedure is
; a list and it has 3 elements.
;- Examples :
; (check-sides? '(6 4 27)) --> evaluates to true
; (check-sides? '(6 0 27)) --> evaluates to false since not all
;                              elements are greater than 0
; (check-sides? '(() 'c 3)) -------> evaluates to false since
;                                    '() and 'c are not numbers
; (check-sides? '(#t 10 14)) -------> evaluates to false since #t is not a number
(define check-sides?
    (lambda (inTriple)
        ...
    )
)
```

```
;- Procedure: sort-all-triples
;- Input : Takes only one parameter named tripleList
;- Output : Returns the list of triples given as the parameter in which
; all triples are sorted internally in ascending order.
;- Hint: You can assume that the given input to this procedure is
; a list of triples (see section 4).
;- Examples :
; (sort-all-triples '((4 3 5) (9 4 6) (13 12 1) (6 6 6))) --> evaluates to
;                                        ((3 4 5) (4 6 9) (1 12 13) (6 6 6))
; (sort-all-triples '((4 7 9) (15 36 9))) --> evaluates to ((4 7 9) (9 15 36))
(define sort-all-triples
        (lambda (tripleList)
          ...
        )
)


;- Procedure: sort-triple
;- Input : Takes only one parameter named inTriple
;- Output : It returns the sorted inTriple in ascending order.
;- Hint: You can assume that the given input to this procedure is
; a triple (see section 3).
;- Examples :
; (sort-triple '(4 3 5)) --> evaluates to (3 4 5)
; (sort-triple '(8 8 8)) -------> evaluates to (8 8 8)
; (sort-triple '(6 10 14)) -------> evaluates to (6 10 14)
(define sort-triple
    (lambda (inTriple)
      ...
    )
)


;- Procedure: filter-triangle
;- Input : Takes only one parameter named tripleList
;- Output : It returns tripleList consists of triples that each triple represents
; a triangle. So, it filters triangles in intripleList and discards other triples.
; The Triangle rule is explained in section 3.
;- Hint: You can assume that the given input to this procedure is
; a list of triples that each of the triples is sorted internally in ascending order.
;- Examples :
; (filter-triangle '((3 4 5) (4 6 9) (1 12 13))) --> evaluates to ((3 4 5) (4 6 9))
; (filter-triangle '((8 10 21) (22 31 53))) --> evaluates to ()
(define filter-triangle
     (lambda (tripleList)
       ...
     ))
```

```
;- Procedure: triangle?
;- Input : Takes only one parameter named triple
;- Output : It returns true if the given triple satisfies the triangle rule,
; returns false if otherwise.
; The Triangle rule is explained in section 3.
;- Hint: You can assume that the given input to this procedure is
; a triple (see section 3) in which all elements are sorted in ascending order.
;- Examples :
; (triangle? '(4 6 8)) ---> evaluates to #t
; (triangle? '(12 21 34)) ---> evaluates to #f
; (triangle? '(9 10 18)) ---> evaluates to #t
(define triangle?
    (lambda (triple)
        ...
    )
)

;- Procedure: filter-pythagorean
;- Input : Takes only one parameter named tripleList
;- Output : It returns tripleList consists of triples that each triple represents
; a pythagorean triangle. So, it filters pythagorean triangles in intripleList
; and discards other triples.
; The Pythagorean theorem is explained in section 3.
;- Hint: You can assume that the given input to this procedure is
; a list of triples that each of the triples is sorted internally in ascending order,
; and satisfies the Triangle rule (see section 3).
;- Examples :
; (filter-pythagorean '((3 4 5) (4 6 8))) ---> evaluates to ((3 4 5))
; (filter-pythagorean '((3 4 5) (13 18 30) (5 12 13) (8 8 8))) ---> evaluates to
 ((3 4 5) (5 12 13))
; (filter-pythagorean '((7 11 16) (9 11 12))) --------> evaluates to ()
(define filter-pythagorean
     (lambda (tripleList)
       ...
     )
)
```

```
;- Procedure: pythagorean-triangle?
;- Input : Takes only one parameter named triple
;- Output : It returns true if the given triple satisfies the Pythagorean theorem,
; returns false if otherwise.
; The Pythagorean theorem is explained in section 3.
;- Hint: You can assume that the given input to this procedure is
; a triple (see section 3) in which all elements are sorted in ascending order.
; Also, the triple itself satisfies the Triangle rule.
;- Examples :
; (pythagorean-triangle? '(4 6 8)) ---> evaluates to #f
; (pythagorean-triangle? '(5 12 13)) ---> evaluates to #t
; (pythagorean-triangle? '(7 24 25)) ---> evaluates to #t
; (pythagorean-triangle? '(9 10 18)) ---> evaluates to #f
(define pythagorean-triangle?
    (lambda (triple)
        ...
    )
)


;- Procedure: sort-area
;- Input : Takes only one parameter named tripleList
;- Output : Returns the list of triples given as the parameter in which
; all triples are sorted according to the areas of the pythagorean triangles
; in ascending order.
;- Hint: You can assume that the given input to this procedure is
; a list of triples that each of the triples is sorted internally in ascending order,
; and satisties the Pythagorean theorem (see section 3).
; Examples :
; (sort-area '((5 12 13) (3 4 5) (16 63 65) (12 35 37))) --> evaluates to
 ((3 4 5) (5 12 13) (12 35 37) (16 63 65))
; (sort-area '((5 12 13) (16 63 65) (3 4 5))) ------>  evaluates to
 ((3 4 5) (5 12 13) (16 63 65))
(define sort-area
      (lambda (tripleList)
       ...
      )
)
```

```
;- Procedure: get-area
;- Input : Takes only one parameter named triple
;- Output : It returns the area of the given pythagorean triangle.
;- Hint: You can assume that the given input to this procedure is
; a triple (see section 3) in which all elements are sorted in ascending order.
; Also, the triple itself satisfies the Pythagorean theorem (see section 3).
;- Examples :
; (get-area '(3 4 5)) ---------> evaluates to 6
; (get-area '(5 12 13)) -------> evaluates to 30
; (get-area '(12 35 37)) ------> evaluates to 210
(define get-area
   (lambda (triple)
    ...
   )
)
```

6 Rules

- You can use let* (or other imperative constructs) for debugging purposes as explained in Section 2. However, you are not allowed to use such features for the actual computation as it provides sequential statement execution. The reason is that, you should get used to the functional way of thinking while using a functional programming language.
- Remove any occurrence of these imperative features before submitting your homework.

7 How to Submit

Submit your Scheme file named as `username-hw4.scm` where `username` is your SU-Course+ username. We will test your submissions in the following manner. A set of test cases will be created to asses the correctness of your scheme procedures. Each test case will be automatically appended to your file. Then the following command will be executed to generate an output. Then your output will be compared against the desired output.

```
scheme < username-hw4.scm
```

So, make sure that the above command is enough to produce the desired output.

8 Notes

- **Important**: Name your files as you are told and **don't zip them.** [-10 points

otherwise]

- **Important: Make sure your procedure name are exactly the same as it is supposed to be!**
- **Important: Since this homework is evaluated automatically make sure your output is exactly as it is supposed to be.**
- No homework will be accepted if it is not submitted using SUCourse+.
- You may get help from our TA or from your friends. However, **you must implement the homework by yourself**.
- Start working on the homework immediately.
- Important, SUCourse's clock may be off a couple of minutes. Take this into account to decide when to submit.
- Note that, you may be able to find Scheme interpreters for Windows. Although it is discouraged, you may use them. However, we want to remind you that, your homework will be evaluated on flow. Hence we recommend that you, at least, test your implementation on flow before submitting.
- Start working on the homework immediately.
- LATE SUBMISSION POLICY:
  Late submission is allowed subject to the following conditions:
  - Your homework grade will be decided by multiplying what you get from the test cases by a "submission time factor (STF)".
  - If you submit on time (i.e. before the deadline), your STF is 1. So, you don't lose anything.
  - If you submit late, you will lose 0.01 of your STF for every 5 mins of delay.
  - We will not accept any homework later than 500 mins after the deadline.
  - SUCourse+'s timestamp will be used for STF computation.
  - If you submit multiple times, the last submission time will be used.