**Sabancı University**
**Faculty of Engineering and Natural Sciences**

**CS305 Programming Languages**

**Homework 5**

Due: May 8, 2020 - Friday @ 23:55

# 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 in the questions of this homework that 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.

---

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

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

1 ]=> (add 'a 4)
;add: 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 also 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

1 ]=>
```

---

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, the `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 "add: actuals are not numbers")))))
;Value: add
```

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

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

2 error>
```

---

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`. You can then have the interpreter interpret the contents of the file. Let us say that we have the file `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 "add: actuals are not numbers")))))

; tests for the procedure add
(add 3 5)
; (add 5 3)
(add 1 2)
; (add 4 'a)
```

---

When you have such a file, you can then have the interpreter process (interpret) the contents of the file in the following way. We normally start the Scheme interpreter

by using the command `scheme`. Instead of this command, simply use the following command: `scheme < add.scm`

Of course, you need to execute this command in the directory in which you have the file `add.scm`

Here is an example for this:

---

```
flow:~> scheme < add.scm
MIT/GNU Scheme running under GNU/Linux
Type ‘^C’ (control-C) followed by ‘H’ to obtain information about interrupts.

Copyright (C) 2014 Massachusetts Institute of Technology
This is free software; see the source for copying conditions. There is NO warranty;
not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Image saved on Saturday May 17, 2014 at 2:39:25 AM
Release 9.2 || Microcode 15.3 || Runtime 15.7 || SF 4.41 || LIAR/x86-64 4.118
Edwin 3.116

1 ]=> ; 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 "add: actuals are not numbers")))))
;Value: add

1 ]=> ; tests for the procedure add
(add 3 5)
;Value: 8

1 ]=> ; (add 5 3)
(add 1 2)
;Value: 3

1 ]=> ; (add 4 'a)
```

```
End of input stream reached.
Moriturus te saluto.
flow:~>
```

---

As you can see, the interpret starts, processes all the expression that we have in the file, and then the interpreter terminates, putting us back the Linux prompt.

# 3 Anapali

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

- A sequence of symbols $\sigma$ is called a *palindrome* if $\sigma$ is the same whether you read it from left to write or from right to left. For example the empty sequence, and the sequences `a`, `aaaa`, `abaaba`, `neveroddoreven` are all palindromes.
- A sequence $\sigma$ is an *anagram* of another sequence $\sigma'$ if the symbols of $\sigma$ can be shuffled to obtain $\sigma'$. For example **abcb** is an anagram of **bacb**, or **aaabbb** is an anagram of **abababab**, **army** is an anagram of **mary**.
- A sequence $\sigma$ is an *anapoli* if $\sigma$ is an anagram of a palindrome. In other words, $\sigma$ is an anapoli if it is possible to rearrange the symbols in $\sigma$ to obtain a palindrome. For example, **prepeellers** is an anapoli since it is an anagram of **lepersrepel** which is a palindrome.

# 4 Scheme procedures to be implemented

You will implement the following 9 procedures in Scheme (in fact, we provide the implementation of the first one)

- `symbol-length`
- `sequence?`
- `same-sequence?`
- `reverse-sequence`
- `palindrome?`
- `member?`
- `remove-member`
- `anagram?`

- `anapoli?`

Below, you will find extensive explanations for these procedures.

We will represent the sequences as lists of Scheme symbols where each symbol will be a single item. For example the palindromic sequence **aba** will be represented as the Scheme list of symbols **(a b a)** with three elements where the first element is the symbol **a**, the second element is the symbol **b** and the third element is again the symbol **a**. We will never use a symbol of length more than one (e.g. a symbol like ab will not be used). In the Scheme procedures to be implemented below, whenever we talk about a "sequence", 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 **symbol-length** defined below 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: symbol-length
;- Input : Takes only one parameter named inSym
;- Output : Returns the number of items in the symbol.
; If the input is not a symbol, then it returns 0.
;- Examples :
; (symbol-length 'a) ------> evaluates to 1
; (symbol-length 'abc) ----> evaluates to 3
; (symbol-length 123) -----> evaluates to 0
; (symbol-length '(a b)) --> evaluates to 0
(define symbol-length
     (lambda (inSym)
          (if (symbol? inSym)
              (string-length (symbol->string inSym))
               0
          )
     )
)

;- Procedure: sequence?
;- Input : Takes only one parameter named inSeq
;- Output : Returns true if inSeq is a list of symbols where each
; symbol is of length 1
;- Examples :
; (sequence? '(a b c)) ----> evaluates to true
; (sequence? '()) ---------> evaluates to true
; (sequence? '(aa b c)) ---> evaluates to false since aa has length 2
; (sequence? '(a 1 c)) ----> evaluates to false since 1 is not a symbol
; (sequence? '(a (b c))) --> evaluates to false since (b c) is not a symbol
(define sequence?
     (lambda (inSeq)
     ...
     )
)
```

```
;- Procedure: same-sequence?
;- Input : Takes two sequences inSeq1 inSeq2 as input
;- Output : Returns true if inSeq1 and inSeq2 are sequences and they are the
; same sequence.
; Returns false if inSeq1 and inSeq2 are sequences but they are
; not the same sequence.
; If inSeq1 is not a sequence and/or inSeq2 is not a sequence,
; it produces an error.
;- Examples :
; (same-sequence? '(a b c) '(a b c)) --> evaluates to true
; (same-sequence? '() '()) ------------> evaluates to true
; (same-sequence? '(a b c) '(b a c)) --> evaluates to false
; (same-sequence? '(a b c) '(a b)) ----> evaluates to false
; (same-sequence? '(aa b) '(a b c)) ---> produces an error
; (same-sequence? '(a b) '(a ba c)) ---> produces an error
(define same-sequence?
        (lambda (inSeq1 inSeq2)
          ...
        )
)

;- Procedure: reverse-sequence
;- Input : Takes only one parameter named inSeq
;- Output : It returns the reverse of inSeq if inSeq is a sequence.
; Otherwise it produces an error.
;- Examples :
; (reverse-sequence '(a b c)) --> evaluates to (c b a)
; (reverse-sequence '()) -------> evaluates to ()
; (reverse-sequence '(aa b)) ---> produces an error
(define reverse-sequence
    (lambda (inSeq)
      ...
    )
)
```

```
;- Procedure: palindrome?
;- Input : Takes only one parameter named inSeq
;- Output : It returns true if inSeq is a sequence and it is a palindrome.
; It returns false if inSeq is a sequence but not a palindrome.
; Otherwise it gives an error.
;- Examples :
; (palindrome? '(a b a)) --> evaluates to true
; (palindrome? '(a a a)) --> evaluates to true
; (palindrome? '()) -------> evaluates to true
; (palindrome? '(a a b)) --> evaluates to false
; (palindrome? '(a 1 a)) --> produces an error
(define palindrome
     (lambda (inSeq)
       ...
     )
)


;- Procedure: member?
;- Input : Takes a symbol named inSym and a sequence named inSeq
;- Output : It returns true if inSeq is a sequence and inSym is a symbol
; and inSym is one of the symbols in inSeq.
; It returns false if inSeq is a sequence and inSym is a symbol
; but inSym is not one of the symbols in inSeq.
; It produces an error if inSeq is not a sequence and/or if
; inSym is not a symbol.
;- Examples :
; (member? 'b '(a b c)) ---> evaluates to true
; (member? 'd '(a b c)) ---> evaluates to false
; (member? 'd '()) --------> evaluates to false
; (member? 5 '(a 5 c)) ----> produces an error
; (member? 'd '(aa b c)) --> produces an error
(define member?
     (lambda (inSym inSeq)
       ...
     )
)
```

```
;- Procedure: remove-member
;- Input : Takes a symbol named inSym and a sequence named inSeq
;- Output : If inSym is a symbol and inSeq is a sequence and inSym is one of
; the symbols in inSeq, then remove-member returns the sequence
; which is the same as the sequence inSeq, where the first
; occurrence of inSym is removed.
; For any other case, it produces an error.
; Examples :
; (remove-member 'b '(a b a b c b)) --> evaluates to (a a b c b)
; (remove-member 'd '(a b c)) --------> produces an error
; (remove-member 'b '()) -------------> produces an error
; (remove-member 'b '(a b 5 c)) ------> produces an error
; (remove-member 5 '(a b c)) ---------> produces an error
(define member?
      (lambda (inSym inSeq)
       ...
      )
)

;- Procedure: anagram?
;- Input : Takes two sequences inSeq1 and inSeq2 as paramaters.
;- Output : If inSeq1 and inSeq2 are both sequences and if inSeq1 is an
; anagram of inSeq2, then anagram? evaluates to true.
; If inSeq1 and inSeq2 are both sequences and but if inSeq1 is not an
; anagram of inSeq2, then anagram? evaluates to false.
; If inSeq1 is not a sequence or if inSeq2 is not a sequence
; then anagram? produces an error.
;- Examples :
; (anagram? '(m a r y) '(a r m y)) --> evaluates to true
; (anagram? '() '()) ----------------> evaluates to true
; (anagram? '(a b b) '(b a a)) ------> evaluates to false
; (anagram? '(a b b) '()) -----------> evaluates to false
; (anagram? '(a bb) '(a bb)) --------> produces an error
; (anagram? 5 '(a b b)) -------------> produces an error
(define anagram?
    (lambda (inSeq1 inSeq2)
       ...
    )
)
```

```
;- Procedure: anapoli?
;- Input : Takes two sequences inSeq1, inSeq2 as parameters.
;- Output : If inSeq1 and/or inSeq2 is not a sequence, it produces
;          an error.
;          When both inSeq1 and inSeq2 are sequences, it returns
;          true if inSeq2 is a palindrome, and inSeq1 is an anagram of
;          the palindrome given by inSeq2.
;          Otherwise, it returns false.
;- Examples :
; (anapoli? '(a b b) '(b a b)) ----------> evaluates to true
; (anapoli? '() '()) --------------------> evaluates to true
; (anapoli? '(d a m a m) '(m a d a m)) --> evaluates to true
; (anapoli? '(a b b c) '(a b c b)) ------> evaluates to false
; (anapoli? '(a b b c) '(a bb bb)) ------> produces an error
; (anapoli? '(a bb) '()) ----------------> produces an error
; (anapoli? 5 '(a))----------------------> produces an error
(define anapoli?
   (lambda (inSeq1 inseq2)
    ...
   )
)
```

# 5 Rules

- You are supposed to use only the expressions, predefined procedures that we covered in the class, except `let*`. You can use let* for debugging purposes as explained in Section 2. However, please remove all `let*` expressions before you submit your homework.

# 6 What to submit

Submit a single file which includes the implementation of the 9 procedures explained above. Your file must be named as:

`username-hw5.scm`

where `username` is your SUCourse username. In order to test your submission, we will start the MIT Scheme interpreter on `flow.sabanciuniv.edu`, and load your file in the interpreter as follows (assuming that your username is abc):

```
1]=> (load "abc-hw5.scm")
;Loading "abc-hw5.scm" -- done
;Value:  ...
```

After loading your file, which will declare the procedures you've implemented in your file, we will try out several examples by using your procedures.

If your file does not load, you will get 0. Therefore, please make sure that you try your file on `flow.sabanciuniv.edu` before you submit it.

# 7 Notes

- **Important:** Name your files as you are told and don't zip them. [**-10 points otherwise**]
- No homework will be accepted if it is not submitted using SUCourse.
- Note that, you may be able to find different Scheme interpreters. 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.sabanciuniv.edu` before submitting.
- Start working on the homework immediately.
- We will have STF (Submission Time Factor) for late submissions (0.01 drop in STF for each 5 mins of delay after the deadline).