Sabancı University
Faculty of Engineering and Natural Sciences

CS305 Programming Languages

Homework 4

Due: 15 December 2021, 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.

Also, you will see that in different parts 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 dificult 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 procedure value add since the declaration of the procedure add is 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 `(add 4 5)`). Hence by using comments, you can test your implementation by applying some test cases.

3 Anapali

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

- A sequence of symbols σ is called a *palindrome* if σ 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 σ is an *anagram* of another sequence σ' if the symbols of σ can be shuffled to obtain σ'. For example **abcb** is an anagram of **bacb**, or **aaabbb** is an anagram of **ababab**, **army** is an anagram of **mary**.
- A sequence σ is an *anapoli* if σ is an anagram of a palindrome. In other words, σ is an anapoli if it is possible to rearrange the symbols in σ 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 procedures in Scheme. 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 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
(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 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 b c b)) --> evaluates to (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 remove-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? 5 '(a bb)) --------------------> produces an error
(define anapoli?
   (lambda (inSeq1 inseq2)
    ...
   )
)
```

5 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.
- Do not use string functions (converting lists to strings etc..) to perform required operations, it is banned!

6 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.

7 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.