Sonoma State University
Computer Science Department
CS 460 – Fall 2017 – Watts

Homework 4

Date Due: 26 October 2017; 11:59 pm. The answers to the questions in the first part of this tutorial should be stored in a file called *yourlastname*H4.txt. The functions you write for Part 2 of this tutorial should be stored in a file called *yourlastname*H4.ss. Both files should be copied to the course dropbox (~tiawatts/cs460drop).

For **Homework 4 Part 1**, you are to complete the following Scheme Tutorial.

This tutorial is designed to give you an introduction to the functional language Scheme. To start the Scheme interpreter on the departmental linux server enter <u>guile</u> at the linux prompt. While running the guile Scheme interpreter commands will be entered at the `guile>` prompt. Enter `ctrl-d` or (`exit`) to exit from the interpreter.

1. Scheme arithmetic expressions are expressed using the standard operators (`+ -* /`) and prefix notation. Enter each of the following Scheme commands at the Guile prompt and record the result of each:

```
(+ 2 5)

(- 5.0 0.5)

(* 5. .5)

(/ 1.5 2.3)

(-(* 2 2/5)(/ 2 5))
```

2. Write and execute Scheme commands to perform each of the following calculations:

```
1.2 × (2 -1/3) + -8.7

(2/3 + 2/9)/(5/1 -2/3)

1 + 1/(2 + 1/(1 + 1/2))

1 × -2 × 3 × -2 × 5 × -6 × 7
```

3. The Scheme language is a descendent of LISP – a list processing language. A literal list is represented as ‘(list of items in list). Enter each of the following commands at the scheme prompt and record the results in your notebook.

```
(car '(a b c))

(cdr '(a b c))
```

```
(cons 'a '(b c))

(cons '(a b) '(c d))

(car '((1 2) (3 4) (5 6)))

(cdr '((1 2) (3 4) (5 6)))

(cons '55 '((1 2) (3 4) (5 6)))

(cons 55 '((1 2) (3 4) (5 6)))

(cons '(5 5) '((1 2) (3 4) (5 6)))

(car '55)

(car 'a)

(cdr (a))

(cdr 'a)

(car (a b c))

(cdr (a b c))

(cons '(b c) 'a)

(cons 5 (a b c))
```

4. Based on the above responses, describe the function of each of the scheme functions represented:

```
car:

cdr:

cons:

the ' symbol:
```

5. Predict the output of the following scheme command:

```
(cons (car '(a b c))
      (cdr '(d e f)))
```

   Enter this command to determine whether your prediction was correct.

Scheme expressions may span more than one line. The Scheme interpreter knows when it has parsed an entire expression by matching double quotes and parentheses.

6. Enter the following Scheme function definition at the prompt:

```
        (define (square n)
                (* n n)
        )
```

The procedure square computes the square of any number n (n^2). Define creates functions and * names the multiplication procedure. Note the form of these expressions. All structured forms are enclosed in parentheses and written in prefix notation, i.e., the operator precedes the arguments. This is true even for simple arithmetic operations such as *.

7. Try using your newly defined square function by entering each of the following commands and recording its result:

```
(square 5)

(square -200)

(square 0.5)

(square -1/2)
```

Note: Scheme systems that do not support exact ratios internally may print 0.25 for `(square -1/2)`.

Exit from the scheme interpreter by entering ctrl-d.

8. Even though the next definition is short, create a file in which it will be stored. Call the file "test.ss."

```
; This function will return the multiplicative reciprocal
; of a numeric input value

(define (reciprocal n)
       (/ 1 n)
)
```

9. Return to Scheme and load your file with the procedure load. (load " yourlastnameH4.ss") Try using the reciprocal procedure you have just defined by entering and recording the result of each of the following commands.

```
(reciprocal 10)

(reciprocal 1/10)

(reciprocal (reciprocal 1/10))

(reciprocal 10.0)

(reciprocal 11.1)

(reciprocal 0)
```

```
(reciprocal ‘abc)
```

10. The last 2 test cases in the previous step should have caused errors. Why? Modify your reciprocal function so that it only returns the reciprocal if the parameter is a non-zero numeric value:

    ```
    ; This function will return the multiplicative reciprocal
    ; of a non-zero numeric input value

    (define (reciprocal n)
            (if (and (number? n) (not (= n 0)))
                    (/ 1 n)
            )
    )
    ```

    Reload the file and rerun the last 2 test cases. What are the results now? Why? Modify the function to display an error message if the reciprocal cannot be calculated:

    ```
    ; This function will return the multiplicative reciprocal
    ; of a non-zero numeric input value

    (define (reciprocal n)
            (if (and (number? n) (not (= n 0)))
                    (/ 1 n)
                    'invalid_parameter
            )
    )
    ```

    What are the results now? What can you surmise about the use of an “if statement” in a Scheme function?

11. Add your square function to your test.ss file. Modify the function so that it will return “invalid_parameter” if the value passed to square is not numeric. Write and test two commands that use both the reciprocal and square functions.

    Command:

    Result:

    Command:

    Result:

12. Write and execute two commands that use simple arithmetic operations and the reciprocal and square procedures. Be creative!

    Command:

    Result:

Command:

Result:

13. Using your book and resources found on the web, define each of the following terms used by the Scheme programming language:

    atom:

    primitive:

    free variable:

    top level definition:

    predicate:

    type predicate:

    lexical scoping:

14. `number?` is a predicate function; it is called to determine if the type of the parameter passed to it is numeric. This function will return `#t` (true) or `#f` (false). Enter each of the following guile commands to explore the functionality of the `number?` function.

```
(number? 5)

(number? '5)

(number? 'a)

(number? '(1 2 3))

(number? (car '(1 2 3)))
```

    Write 2 more commands that you expect to return #t:

    Write 2 more commands that you expect to return #f:

    Were your predictions accurate?

    What conclusions have you drawn about the functionality of the `number?` function?

15. `symbol?` Is also a predicate function. Enter each of the following guile commands to explore the functionality of the `symbol?` function.

```
(symbol? 5)

(symbol? '5)
```

```
(symbol? 'a)

(symbol? '(1 2 3))

(symbol? (car '(1 2 3)))
```

Write 2 more commands that you expect to return #t:

Write 2 more commands that you expect to return #f:

Were your predictions accurate?

What conclusions have you drawn about the functionality of the `symbol?` function?

16. Similarly, explore the functionality of the `list?` function.

    What conclusions have you drawn about the functionality of the `list?` function?

17. Similarly, explore the functionality of the `zero?` function.

    What conclusions have you drawn about the functionality of the `zero?` function?

18. Similarly, explore the functionality of the `null?` function.

    What conclusions have you drawn about the functionality of the `null?` function?

19. Similarly, explore the functionality of the `char?` function.

    What conclusions have you drawn about the functionality of the `char?` function?

20. Similarly, explore the functionality of the `string?` function.

    What conclusions have you drawn about the functionality of the `string?` function?

21. Enter the following Scheme code into a file called cond.ss:

```
(define (cond_ex_1 p)
        (cond ((= p 0) 'equal)
              ((< p 0) 'negative)
              ('positive)
        )
)

(define (cond_ex_2 param)
        (cond ((= param 1) 'The_value_is_1)
              ((= param 2) 'The_value_is_2)
              ((= param 12) 'The_value_is_12)
              ('none_of_the_above)
        )
)
```

```
(display (cond_ex_1 5))
(newline)
(display (cond_ex_1 -5))
(newline)
(display (cond_ex_1 0))
(newline)
(display (cond_ex_2 1))
(newline)
(display (cond_ex_2 12))
(newline)
(display (cond_ex_2 5))
(newline)
(display (cond_ex_2 2))
(newline)
(display (cond_ex_1 -5))
(newline)
(display (cond_ex_1 0))
(newline)
(display (cond_ex_2 1))
(newline)
(display (cond_ex_2 12))
(newline)
(display (cond_ex_2 5))
(newline)
(display (cond_ex_2 2))
(newline)
```

22. Interpret the code in this file by entering:

    ```
    guile cond.ss
    ```

    at the Linux prompt.

    What are the results?

23. Add a third function, called cond_ex_3 to your cond.ss file. This function should have 2 parameters: choice and value. If choice is 1, the function should return value. If choice is 2, the function should return the square of value. If choice is 3, the function should return the reciprocal of the value. Otherwise the function should return 0.

    Add display commands to test cond_ex_3 and test your newly added function.

24. Can a `cond` function call be translated to a C++ switch statement? Why or why not?

25. Create a file called lists.ss containing the following functions:

```
(define (list_copy1 ls)
        (if (list? ls)
                ls
                'list_copy1_requires_a_list_argument
        )
)
```

```
(define (list_copy2 ls)
        (if (list? ls)
                (if (null? ls)
                        '()
                        (cons (car ls) (list_copy2 (cdr ls)))
                )
                'list_copy2_requires_a_list_argument
        )
)
```

26. Add commands to the file to test these functions.

27. `list_copy1` and `list_copy2` should function identically. However, `list_copy2` illustrates a recursive approach to parsing a list and accessing the elements of the list. In `list_copy2`, modify the line:
    ```
    (cons (car ls) (list_copy2 (cdr ls)))
    ```
    to:
    ```
    (cons (car ls) (list_copy2 (cddr ls)))
    ```

    How does this change the functionality of list_copy2?

For **Homework 4 Part 2**, you are to write the following Scheme functions. To write these functions, you should only use the Scheme primitives introduced in the Tutorial. Place the well documented implementations of these functions in a file called *yourlastname*H4.ss

1. Copy the functions `square` and `reciprocal` to your file.

2. Copy the functions `list_copy1` and `list_copy2` to your file. Rename `list_copy1` as `list_copy`. Rename `list_copy2` as `odd_copy`.

3. Write and test a recursive Scheme function called even_copy that will create a copy of the elements in the even numbered positions in a list starting with the second element in the list. The formal argument should be a list.
    ```
    (define (even_copy mylist)
         . . .
    )
    ```

4. Write and test a Scheme function called `list_sum` that will return the sum of the numeric values in the list. This function should use recursion to calculate the sum. The formal argument should be a list. If the argument is not a list or if there are no numeric values in the list, this function should return 0.
    ```
    (define (list_sum mylist)
         . . .
    )
    ```

5. The scheme command `car` returns the first value in a list. Write and test a Scheme function called last that will return the last item in an existing list. The formal argument should be a list.
    ```
    (define (last mylist)
         . . .
    )
    ```

6. The scheme command `cons` inserts an item as the first item in an existing list. Write and test a Scheme function called insert_last that will insert a value into a list as the last element of the list. The formal arguments should be the value to be inserted into the list and the list.

```
   (define (insert_last myvalue mylist)
         . . .
   )
```

7. The scheme command `cdr` returns a list after the first value has been removed. Write and test a Scheme function called remove _first that will remove the first element of a list and return the resulting list. The formal argument should be a list.

```
   (define (remove_first mylist)
         . . .
   )
```

8. The scheme command `cdr` returns a list after the first value has been removed. Write and test a Scheme function called remove _last that will remove the last element of a list and return the resulting list. The formal argument should be a list.

```
   (define (remove_last mylist)
         . . .
   )
```

9. Write and test a Scheme function called list_reverse that will reverse the elements of a list. The formal argument should be a list.

```
   (define (list_reverse mylist)
         . . .
   )
```

10. Write and test a Scheme function called sqrt. This function should calculate the square root of a numeric value to 5 decimal places of precision. The formal argument should be a non-negative (0 or positive) numeric value.

```
(define (sqrt n)
   . . .
)
```