

# Scheme Tutorial Exercise 1 (Introduction to Scheme)

Topics: Beginning Scheme

Goals: Upon successful completion of this tutorial you should be able to:

1. Define basic Scheme terms
2. Manipulate a list using Scheme primitives
3. Define Scheme functions

Related text sections:

## Scheme Tutorial Exercise 1 Instructions

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 guile 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.

Enter each of the following commands at the scheme prompt and record the result.

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

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

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

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

car:

cdr:

cons:

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.

As you can see, Scheme expressions may span more than one line. The Scheme system knows when it has an entire expression by matching double quotes and parentheses.

Enter the following Scheme procedure definition:

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

The procedure `square` computes the square  $n^2$  of any number  $n$ . `Define` establishes variable bindings, `lambda` creates procedures, 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. As you can see, this is true even for simple arithmetic operations such as `*`.

Try using `square` 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).

Even though the next definition is short, please enter it using a file. Exit from the scheme interpreter by entering ctrl-d. Call the file "reciprocal.ss."

```
(define reciprocal
  (lambda (n)
    (if (= n 0)
        "oops!"
        (/ 1 n))))
```

This procedure, reciprocal, computes the quantity  $1/n$  for any number  $n$ . For  $n = 0$ , reciprocal returns the string "oops!". Return to Scheme and load your file with the procedure load.

```
(load "reciprocal.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 0)
```

```
(reciprocal (reciprocal 1/10))
```

Write and execute two commands that use both the reciprocal and square procedures (Note: you will need to redefine the square procedure to test your commands.) Be creative!

1. Command:

Result:

2. Command:

Result:

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

```
(+ 4 5)
```

```
(* 1.5 2.3)
```

```
(- (* 4 4/5) (/ 4 5))
```

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

$$1.2 \times (2 - 1/3) + -8.7$$

$$(2/3 + 4/9)/(5/1 - 4/3)$$

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

$$1 \times -2 \times 3 \times -4 \times 5 \times -6 \times 7$$

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

1. Command:

Result:

2. Command:

Result:

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

atom:

primitive:

lambda expression:

free variable:

top level definition:

predicate:

type predicate:

lexical scoping:

## Scheme Tutorial Exercise 2 (Scheme Lists)

Topics: Advanced Scheme functions

Goals: Upon successful completion of this tutorial you should be able to:

1. Design and implement Scheme functions to manipulate the items in a list

Related text sections:

## Scheme Tutorial Exercise 2 Instructions

1. Describe the function and use of each of the following Scheme primitive functions:
  - a. list?
  - b. number?
  - c. null?
  - d. caddr
  - e. if
  - f. let
2. Write and test a Scheme function called insert-first that will insert a value into a list as the first element of the list. The formal arguments should be a list and the value to be inserted into the list.
3. Write and test a Scheme function called remove-first that will remove the first element of a list. The formal argument should be a list.
4. What happens when you enter (insert-first 1 2)? What happens when you enter (remove-first 1)? Modify the functions you wrote in steps 1 and 2 to test their input values to prevent these problems. If the input values are incorrect, your functions should issue an informative error message.
5. What happens if you enter (remove-first '())? Modify your remove-first function to test its input value to prevent this problem. If the input list is empty, your function should issue an informative error message.
6.
  - a. Write and test a recursive Scheme function called list-copy that will create a copy of a list. The formal argument should be a list.
  - b. Write and test a recursive Scheme function called odd-copy that will create a copy of the elements in the odd numbered positions in a list starting with the first element in the list. The formal argument should be a list.
  - c. 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.
7. 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 a list and the value to be inserted into the list.
8. Write and test a Scheme function called remove-last that will remove the last element of a list. The formal argument should be a list.
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.

## Scheme Tutorial Exercise 3 (Scheme Sorting)

Topics: List sorting using Scheme

Goals: Upon successful completion of this tutorial you should be able to:

1. Design and implement Scheme functions to recursively sort the items in a list

Related text sections:

### Scheme Tutorial Exercise 3 Instructions

1. Execute the list reversing procedure you wrote for Tutorial 2 using '(a (b c) d) as input. Does it return (d (c b) a)? Or does it return (d (b c) a)? Write a new procedure called all-reverse that recursively reverses nested lists. Execute your new procedure using '(a (b (c d)) (e f)) as input. It should return ((f e) ((d c) b) a).
2. Each of the recursive sorts we have discussed and implemented are of the form:

```
recursively-sort (list L)
{
  if (size of L > 1)
  {
    split L into 2 parts: L1 and L2
    recursively-sort L1
    recursively-sort L2
    combine sorted lists L1 and L2 into a sorted list L
  }
}
```

2. a. Write a scheme procedure called quicksort that uses Quick Sort to sort its single list argument into **ascending** order. You will probably need to create a helper function (or functions) to split the list into 2 parts – those elements < than the pivot and those >= to the pivot. You may also need an append procedure.

OR

2. b. Write a scheme procedure called mergesort that uses Merge Sort to sort its single list argument into **descending** order. You will probably need to create a helper function (or functions) to split the list into 2 parts. You may also need a merge procedure.
3. Run each of your sorts using the list '(1 5 3 6 8 92 -1 0 4 5 3). Run each of your sorts using the list '(a b d e c t s). How do your sorts work with these input lists? Modify your sorts so that they sort only lists of numeric values.
4. Place the well documented procedures you created for Scheme tutorials 2 and 3 in a file called *your-last-name.ss* and drop the file into the cs460drop folder.