CSE1729 - Introduction to Programming September 4, 2019

Lab Assignment 1

Objectives

1. Understand arithmetic expression and operators
2. Write some simple functions in SCHEME
3. Be able to submit your work via the MIMIR site for CSE1729
   (use the link to Mimir found on the Moodle site here: `http://courses.engr.uconn.edu`).

Starting Racket

We again encourage you to use `DrRacket` on your machine to get familiar with the environment and see how it can help you debug your programs. While Mimir is great for grading and immediate feedback, it does not offer a lot of support for debugging.

`DrRacket` runs a REPL (*read-eval-print*) loop. That is, it continuously repeats a cycle of reading a SCHEME expression, evaluating that expression and printing the result of evaluating that expression. Right now it is just waiting for an expression to evaluate. Since Racket supports several dialects of Scheme, we can indicate the particular dialect we want the interpreter to emulate. For most of the semester, we will follow the R5RS standard which we can indicate opening the Language -> Choose Language menu item and selecting the “Other Languages” radio button and clicking on R5RS in the dialog box and clicking on the “OK” button. Go ahead and do that now. You should use this mode unless otherwise indicated for the rest of the course.

Lab Assignment Activities

1. Currency Conversion
   (a) The exchange rate between U.S. dollars and euros is 1\$ = 0.91 €. Write a SCHEME procedure, named `usd-to-euro`, to convert dollars into euros. How many euros will you get when you exchange \$250?
   (b) The exchange rate between euros and Japanese yen is 1 € = 116.77 ¥. Write a SCHEME procedure, named `euro-to-yen`, to convert euros into yen. How many yen will you get when you exchange € 250?
   (c) Write a procedure, named `usd-to-yen`, using your solutions from parts (a) and (b) to convert U.S. dollars to yen. How many yen would you receive in exchange for \$250?
2. (a) Define a variable representing the mathematical constant e. You must name your constant `e` and use the approximation 2.71828 as the value for your variable.

(b) Define a function `tanh` which, given a positive number x, returns the hyperbolic tangent of x defined as

$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}.$$

You may use the built in SCHEME function `(expt b e)` function to complete this problem. `(expt b e)` computes b^e.

3. Matrices

A *matrix* is a rectangular grid of numbers organized into rows and columns. Matrices are an important tool in algebra and are often used to solve systems of linear equations. Below are examples of a couple of 2×2 matrices (matrices with 2 rows and 2 columns) that we will call M and N.

$$M = \begin{pmatrix} 2 & -4 \\ -6 & 12 \end{pmatrix} \qquad N = \begin{pmatrix} -3 & 1 \\ 2 & 7 \end{pmatrix}$$

(a) A special value associated with any 2×2 matrix is the *determinant*. Given a generic 2×2 matrix, the determinant can be computed using the following formula:

$$\det\begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$$

Using the formula, we can compute the determinant of matrix M above as $(2)(12)-(-4)(-6) = 0$. Write a Scheme procedure to compute the determinant of a generic 2×2 matrix. Assume that the matrix elements a. b, c and d are given as four formal parameters. Compute the determinant of N.

(b) A matrix is called *invertible* if its determinant is non-zero. Write a procedure that checks whether or not a generic 2×2 matrix is invertible. Verify that N is invertible and M is not invertible.

(c) A powerful property of matrices is that certain kinds of matrices may be meaningfully *multiplied* together to get another matrix. (It turns out that matrix multiplication is intimately related to composition of linear functions, but you won't need this interpretation to complete the exercise.) In particular, it is possible to multiply 2×2 matrices. Assume we have two matrices:

$$A = \begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} \qquad B = \begin{pmatrix} a_2 & b_2 \\ c_2 & d_2 \end{pmatrix}$$

The product of these matrices is defined to be

$$A \cdot B = \begin{pmatrix} a_1a_2 + b_1c_2 & a_1b_2 + b_1d_2 \\ c_1a_2 + d_1c_2 & c_1b_2 + d_1d_2 \end{pmatrix}.$$

Given two 2×2 matrices, we wish to determine whether or not their product $A \cdot B$ will be invertible. There are two ways to do this

i. Compute the product, as described above; then compute its determinant. Define a function named `(prod-inv-direct? a1 b1 c1 d1 a2 b2 c2 d2)` which determines if the product of two matrices is invertible by this method.

ii. It is a remarkable fact that for two matrices A and B, $\det(A \cdot B) = \det(A) \times \det(B)$. Thus, we can compute the determinant of $A \cdot B$ indirectly (without computing the product of the two matrices) from the determinants of A and B. Define a function named `(prod-inv-indirect? a1 b1 c1 d1 a2 b2 c2 d2)` which determines if the product of two matrices is invertible by this method.

Once you have finished:

1. Save your work (the definitions) to a file named lab1.rkt
2. Submit your lab solution file for grading via Mimir.
3. If you haven't already, read the Honor Code Pledge on the CSE1729 Moodle site completely, sign it and return it..

Please note: assignments will not be graded for credit until your Honor Code Agreement is filed-see it under Course Content in Moodle.

CSE1729 - Introduction to Programming September 10, 2019

Laboratory Assignment 2

Objectives

- Work with recursive functions
- Work with the conditional `if` and `cond` special forms.

Activities

1. Write a recursive function, named `(zeno n)`, that computes the sum of the first n terms of the following series from Zeno's Dichotomy Paradox, $Z_n = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^n}$. Use the SCHEME built-in function `(expt b e)` function to compute the denominator of each term.

2. **Number of Digits** Write a SCHEME procedure, named `(num-digits n)`, which accepts an integer `n` and returns the number of digits in `n`. For example, both `(num-digits 10000)` and `(num-digits 12345)` evaluate to 5.

3. Young Jeanie knows she has two parents, four grandparents, eight great grandparents, and so on.
   (a) Write a recursive function, named `(a n)` to compute the number of Jeanie's ancestors in the n^{th} previous generation. The number of ancestors in each generation back produces a sequence that may look familiar:
   $$2, 4, 8, 16, \ldots$$
   For each generation back, there are twice the number of ancestors than in the previous generation back. That is, $a_n = 2a_{n-1}$. Of course, Jeanie knows she has two ancestors, her parents, one generation back.
   (b) Write a recursive function to compute Jeanie's total number of ancestors if we go back n generations. Specifically, `(num-ancestors n)` should return:
   $$2 + 4 + 8 + \cdots + a_n$$
   Use your function in part (a) as a "helper" function in the definition of `(num-ancestors n)`[1].

4. **Combinations:** Often, we are interested in the number of ways a group of k objects can be formed from a total of n objects. For instance, if I needed to select three students from a lab section containing twenty students. How many ways are there to make this selection? First, notice that for the first student selected, I can choose any one out of the twenty in the section. Therefore, there are twenty ways to make this selection. For the second student, I am selecting from the nineteen remaining students. Therefore, there are nineteen ways to make this selection. Finally, I am making the selection of the last student from the remaining eighteen. So, there are eighteen ways to make the final selection. But, the ordering of our selections don't make any difference. Consider the number of ways we can select a group of three students A, B and C. ABC, ACB, BAC, BCA, CBA and CAB give us six ways the same group of three have been counted. So, the total number of groups that can be counted is:
   $$\frac{20 \cdot 19 \cdot 18}{3 \cdot 2 \cdot 1}$$

[1] Of course, we can use the closed-form solution for the geometric progression to compute `num-ancestors` ($ancestors(n) = 2^{n+1} - 2$) but that doesn't give us any experience with recursive functions. However, this is a useful fact we can use when testing our functions to ensure they are correct.

In general, we use the following equation to compute the number of groups of size k that can be formed from n objects:

$$\frac{n!}{(n-k)!k!}$$

You will notice that the $(n-k)!$ term in the denominator cancels the last $(n-k)$ terms in the product of $n!$ in the numerator, leaving just the product of the top k terms from $n!$. Also notice, the $k!$ term prevents us from counting the duplicate ways of selecting the same k objects. This computation has it's own terminology and notation. The notation is $\binom{n}{k}$ and is read "n choose k" indicating this quantity represents the number of ways of choosing k objects from a group of n objects. So,

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

Use the `factorial` function discussed in lecture to write an `(n-choose-k n k)` function that takes two parameters, n and k and computes the value of $\binom{n}{k}$ shown above.

Remark 1. *Note that* `(n-choose-k n k)` *should evaluate to zero when $n < k$ or $k < 0$.*

5. Interestingly, $\binom{n}{k}$ is the binomial coefficient indexed by n and k in the polynomial expansion of $(x+y)^n$. That is, by the binomial theorem, the polynomial expansion of $(x+y)^n$ is the sum of ax^by^c terms where a is specified by $\binom{n}{b}$ or $\binom{n}{c}$ which are equal. These binomial coefficients also form the sequence of natural numbers in Pascal's Triangle which has interesting properties and contains many patterns.

   The cells in Pascals Triangle, pictured below in Figures 1 and 2, are arranged by a row number, starting at the top at row $n = 0$, and an entry number starting on the left of each row at entry $k = 0$. The entry in row n=0, is defined as 1. Figure 1 shows the second entry in row 2 computed as $\binom{1}{0} + \binom{1}{1} = 1 + 1 = 2$. In general, the entries in Pascal's Triangle are determined by the recurrence relation referred to as Pascal's Law:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

   Define a SCHEME function, named `(pascals-triangle n k)`, which computes the entry in Pascal's Triangle at row n and entry k recursively, without factorial, by using Pascal's Rule defined above. Take great care with your base case(s). That is, be sure the entries you are adding from the previous row exist. Figure 2 shows an example of the first five rows of Pascal's Triangle.

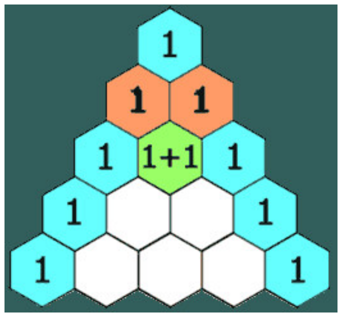


Figure 1: Illustration of $\binom{3}{2}$ decomposed into the sum $\binom{1}{0} + \binom{1}{1}$ arranged in Pascal's Triangle.

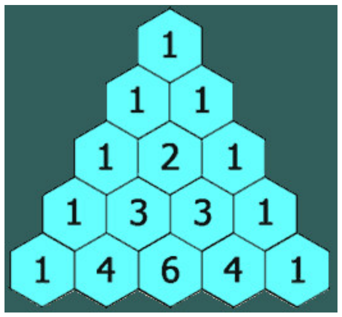


Figure 2: Five rows completed in Pascal's Triangle.

CSE1729 - Introduction to Programming September 15, 2019

Laboratory Assignment 3

Activities

1. (a) The Pell numbers are an infinite sequence of integers which correspond to the denominators of the closest rational approximations of $\sqrt{2}$. The Pell numbers are defined by the following recurrence relation (which looks very similar to the Fibonnacci sequence):

$$P_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ 2P_{n-1} + P_{n-2} & \text{otherwise} \end{cases}$$

Use this recurrence relation to write a recursive function, `pell`, which takes one parameter, n, and returns the n^{th} Pell number.

   (b) Write a separate function, named `(find-pell n)`, which uses your Pell function to find the largest Pell number which is less than n. You can try testing it by finding the largest Pell number less than 100.

   (c) The numerator for the rational approximation of $\sqrt{2}$ corresponding to a particular Pell number is half of the corresponding number in the sequence referred to as the *companion Pell numbers* (or Pell-Lucas numbers). The companion Pell numbers are defined by the recurrence relation:

$$Q_n = \begin{cases} 2 & \text{if } n = 0 \\ 2 & \text{if } n = 1 \\ 2Q_{n-1} + Q_{n-2} & \text{otherwise} \end{cases}$$

Use this recurrence relation to write a function, named `(comp-pell n)`, which returns the n^{th} companion Pell number.

   (d) Finally write a function that uses the Pell number and companion Pell number functions, as described in Part a, to write a SCHEME function, named `(sqrt-2-approx n)`, to compute the n^{th} approximation for $\sqrt{2}$. You can test your new function to compute the approximation for $\sqrt{2}$ using the sixth Pell and companion Pell numbers.

2. Viète's formula is the following infinite product of nested radicals representing the mathematical constant π:

$$\frac{2}{\pi} = \frac{\sqrt{2}}{2} \cdot \frac{\sqrt{2+\sqrt{2}}}{2} \cdot \frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2} \cdots$$

Write a SCHEME function named `(viete n)` which approximates $\frac{2}{\pi}$ using Viète's formula.
Note: You may want to use a helper function that takes an additonal parameter so that you can compute the numerator of each term easily (in a `let` form perhaps) as well as pass it to the recursive call.

3. It is an interesting fact the the square-root of any number may be expressed as a *continued fraction.* For example,

$$\sqrt{x} = 1 + \cfrac{x-1}{2 + \cfrac{x-1}{2 + \cfrac{x-1}{\ddots}}}$$

Write a Scheme function called *new-sqrt* which takes two formal parameters x and n, where x is the number we wish to find the square root of and n is the number of continued fractions to compute recursively. Demonstrate that for large n, *new-sqrt* is very close to the builtin *sqrt* function.

4. **Nested Recursion**:The McCarthy 91 function is a recursive function, defined by the computer scientist John McCarthy as a test case for formal verification within computer science.

   The McCarthy 91 function is defined as

$$m91(n) = \begin{cases} n - 10, & \text{if } n > 100 \\ m91(m91(n+11)), & \text{if } n \leq 100 \end{cases}$$

   Define a SCHEME procedure, named `(m91 n)`, which evaluates to McCarthy's 91 function. Try a few values for n which are less than 100 and a few greater than 100.

CSE1729 - Introduction to Programming September 24, 2019

Laboratory Assignment 4

Activities

1. **Using the special form** `let`, we can declare some substitutions for use within an expression. To illustrate this, we use an equation relating atmospheric pressure p to altitude h and other parameters:

$$p = p_0 \cdot (1 - \frac{L \cdot h}{T_0})^{(\frac{g \cdot M}{R \cdot L})}$$

This function has several constant parameters as detailed in the table below. For a more detailed introduction, see `http://en.wikipedia.org/wiki/Atmospheric\_pressure\#Altitude\_atmospheric\_pressure\_variation`.

Parameter	Description	Value
p_0	sea level standard atmospheric pressure	101325 Pa
L	temperature lapse rate	0.0065 K/m
T_0	sea level standard temperature	288.15 K
g	Earth-surface gravitational acceleration	9.80665 m/s^2
M	molar mass of dry air	0.0289644 kg/mol
R	universal gas constant	8.31447 $J/(mol \cdot K)$

For our own convenience, you can define these constants within the function for this equation, establishing local bindings the interpreter can use during evaluation of the function. In order to make the expression in the function body easier to read, we can also add bindings for the base and exponent portions of the body expression. To do this, we must use nested `let` statements here since none of the variables are bound to values until the entire list has been evaluated. Define a SCHEME function, named `(pressure h)`, using the `let` form to simplify the function body.

2. **The Tower of Hanoi** is a classic problem often given to students learning about recursion. The puzzle was invented by mathematician Édouard Lucas (of Lucas Numbers fame) in 1883.
The problem consists of three pegs onto which are placed disks of different sizes. At the start, all disks are stacked onto one peg, with each placed onto a larger disk (see Figure 1). Let's call that the source peg. The object of the game is to move the entire stack of disks from the source peg to the destination peg. You may use the remaining peg, the temporary peg, to hold disks while you are in the process of moving disks to the destination peg. However, you may only move one disk at a time and you may not place a disk on top of a smaller disk.

At first, it may not be obvious how to develop a recursive decomposition for this problem. The key observation is, if you could recursively move the stack of all but the largest disk to the temporary peg (see Figure 2), you could then move the largest disk to the destination peg (see Figure 3. Once the largest disk is on the destination peg, you can then, again recursively, move the stack of the remaining disks on top of the largest disk at the destination peg (see Figure 4).

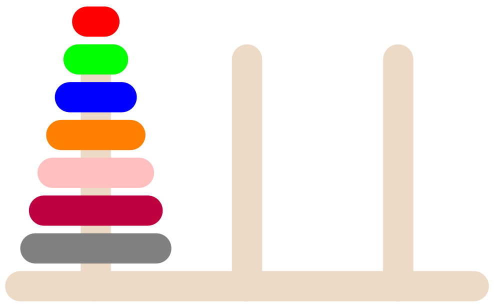


Figure 1: Starting configuration for Tower of Hanoi.

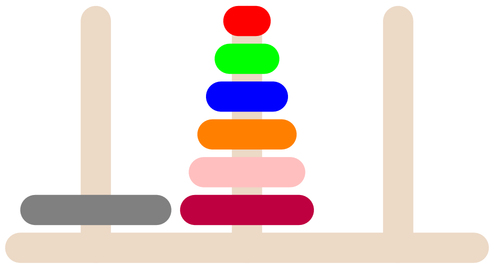


Figure 2: Step 1: Move the stack of all except the largest disk to the temporary peg.

For this problem, define a SCHEME procedure, named `(tower n source temp dest)`, which gives *step-by-step instructions* on how to move a stack of n disks from the `source` peg to the `dest` peg, using the `temp` peg to temporarily hold smaller disks while larger disks are moved into place.
In order to print instructions to the interactions window of the Racket IDE, you will need to use the SCHEME `display` function. You should use the following commands to print the instructions to move the top disk from one peg to another:

```
(display "Move the disk from peg ")
(display source)
(display " to peg " )
(display dest)
(newline))
```

You must copy these expressions precisely to pass the Mimir test cases. Also, use a `cond` form for your conditional expression instead of an `if` form.

See below for sample output from an evaluation of the `tower` function using four disks and moving the stack from peg 1 to peg 3:

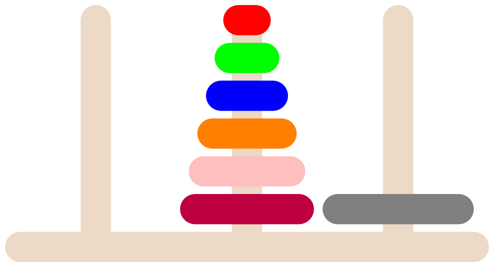


Figure 3: Step 2: Move the largest disk to the destination peg.

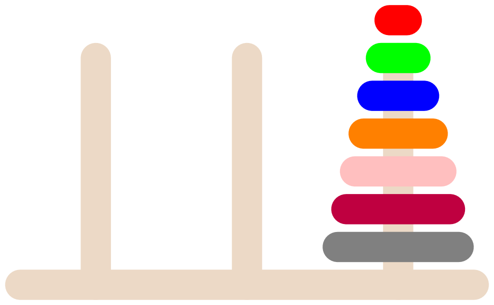


Figure 4: Step 3: Move the stack of all except the largest disk to the destination peg.

```
> (towers-of-hanoi 4 1 2 3)
Move the disk from peg 1 to peg 2
Move the disk from peg 1 to peg 3
Move the disk from peg 2 to peg 3
Move the disk from peg 1 to peg 2
Move the disk from peg 3 to peg 1
Move the disk from peg 3 to peg 2
Move the disk from peg 1 to peg 2
Move the disk from peg 1 to peg 3
Move the disk from peg 2 to peg 3
Move the disk from peg 2 to peg 1
Move the disk from peg 3 to peg 1
Move the disk from peg 2 to peg 3
Move the disk from peg 1 to peg 2
Move the disk from peg 1 to peg 3
Move the disk from peg 2 to peg 3
```

3. **SICP Exercise 1.42** Let f and g be two one-argument functions. The composition f after g is defined to be the function $x \mapsto f(g(x))$. Define a procedure, named `(compose f g)`, that implements composition. For example, if `inc` is a procedure that adds 1 to its argument,

   ```
   ((compose square inc) 6)
   49
   ```

   Note: Your procedure should return a *function* that performs the composition of f and g.

4. **SICP Exercise 1.43** If f is a numerical function and n is a positive integer, then we can form the n^{th} repeated application of f, which is defined to be the function whose value at x is $f(f(...(f(x))...))$. For example, if f is the function $x \mapsto x+1$, then the n^{th} repeated application of f is the function $x \mapsto x+n$. If f is the operation of squaring a number, then the n^{th} repeated application of f is the function that raises its argument to the 2^n th power. Write a procedure named `(repeated f n)`, that takes as inputs a procedure that computes f and a positive integer n and returns the procedure that computes the n^{th} repeated application of f. Your procedure should be able to be used as follows:

   ```
   ((repeated square 2) 5)
   625
   ```

   You may want to take advantage of the `compose` function you wrote.

5. Recall the McCarthy 91 function from Lab 3. Consider the following form for the 91 function:

   $$f(x) = \begin{cases} x - 10, & \text{if } x > 100 \\ f^{91}(x+901), & \text{if } x \leq 100 \end{cases}$$

   where $f^{91}(y)$ stands for $f(f(\cdots f(y)\cdots))$, the 91-times-repeated application of f. That is, f composed with itself 90 times. Write a SCHEME function, named `(m91 x)` which computes the 91 function as defined above. That is, use the `repeated` function you wrote earlier to compute the `m91` function.

CSE1729 - Introduction to Programming October 14, 2019

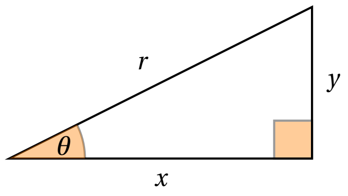
Laboratory Assignment 5

Objectives

Work with pairs and lists

Activities

1. There are two main systems of defining points on a two-dimensional plane. One consists of a distance from the origin and an angle from the positive x-axis, referred to as polar coordinates. The other, more familiar system, consists of two components corresponding to the distance along the x-axis and the distance along the y-axis from the origin, referred to as Cartesian coordinates.



   (a) Converting to polar coordinates: Define a SCHEME function named `(c->p p)` which accepts a point in the Cartesian coordinate system as a *pair* and returns another pair representing the same point in the polar coordinate system. That is, if the function receives the pair `(x . y)` as a parameter, it should evaluate to the pair `(r . θ)`, where $r = \sqrt{x^2 + y^2}$ and $\theta = tan^{-1}(\frac{y}{x})$. Your function should take a SCHEME pair as a parameter and return a SCHEME pair. Note: the arctan function in SCHEME is named `atan`.

   (b) Converting to Cartesian coordinates: Define a SCHEME function named `(p->c p)` which accepts a point in the polar coordinate system as a *pair* and returns another pair representing the same point in the Cartesian coordinate system. That is, if the function receives the pair `( r . θ)` as a parameter, it should evaluate to the pair `(x . y)`, where $x = r \cdot cos(\theta)$ and $y = r \cdot sin(\theta)$. Your function should take a SCHEME pair as a parameter and return a SCHEME pair.

2. You may recall that, given two points, (x_1, y_1) and (x_2, y_2), one can find the slope, m, of a straight line through these two points with the equation:
$$m = \frac{y_2 - y_1}{x_2 - x_1}$$
Furthermore, the function that defines a straight line in "slope intercept form" has the form $y = mx + b$ where b is the *y-intercept*. Given the slope, m, of a line and a point, (x_1, y_1), on the line, one can find the y-intercept by the equation $b = y_1 - mx_1$.
Define a SCHEME function, named `(y p1 p2)`, that takes two points, `p1` and `p2` (each point stored in a SCHEME pair), as parameters and evaluates to a function of one parameter that, given x, will return the corresponding y for a point on the straight line between `p1` and `p2`.
Hint: If you want to define variables for m and b, using the variable m in the expression that defines b, you can avoid using nested `let` *forms by using* `let*` *instead.*

3. **Hamming Weight** The Hamming weight, named after the computer scientist Richard W. Hamming, of a string is the number of symbols that are different from the zero-symbol of the alphabet used. Define a SCHEME function, named `(hamming-weight lst)`, which takes a list of integers and returns the number of non-zero integers in the list.

4. **Hamming Distance** In information theory, the Hamming distance function, also named after the computer scientist Richard W. Hamming, gives a measure of the "difference" between two strings that have the same length. In other words, it measures the minimum number of substitutions required to change one string into the other, or the minimum number of errors that could have transformed one string into the other. Define a SCHEME function, named `(hamming-distance l1 l2)`, which takes two lists of integers of equal length as parameters, and returns the number of positions in which the value in `l1` and the value in `l2` are different.

5. **Scalar-Vector Multiplication** Multiplying a vector by a scalar produces a vector where each component is the corresponding value in the original vector multiplied by the scalar quantity. For example:

$$a(x_1 \quad x_2 \quad x_3) = (ax_1 \quad ax_2 \quad ax_3)$$

Define a SCHEME function, named `sv-mult`, which takes a list and a value as parameters and performs scalar-vector multiplication on them.

6. **Vector Addition** Adding two vectors produces a vector where each component is the sum of the corresponding values in the original vectors. For example:

$$(x_1 \quad x_2 \quad x_3) + (y_1 \quad y_2 \quad y_3) = (x_1 + y_1 \quad x_2 + y_2 \quad x_3 + y_3)$$

Define a SCHEME function, named `v-add`, which takes two lists and performs vector addition on them. *Note: **vector subtraction** is structured in the same way.*

7. The **dot product** of two lists of numbers $(x1\ x2\ x3)$ and $(y1\ y2\ y3)$ is

$$x1 * y1 + x2 * y2 + x3 * y3$$

Define a recursive SCHEME function `(dot x y)` that takes two lists of numbers as its inputs and returns the dot product of those two lists. Do not use the in-built `map` function. You can assume the two lists have the same length.

CSE1729 - Introduction to Programming October 21, 2019

Laboratory Assignment 6

Objectives

More work with lists

Activities

1. The median value of a set of n numbers is the value that separates the half of higher values from the half of lower values in the set. The median can be found by arranging the values in the set in (sorted) order and choosing the "middle" value. If there are an even number of values in the set, the median is described as the mean of the two middle values.
   (a) Define a SCHEME function named `(list-at l i)` which, given a list `l` and an index `i` returns the element in the list at index `i`. You should consider the first element of the original list to be at index 0.
   (b) Write a SCHEME function, named `list-median`, that takes a list of numbers as a parameter and returns the median value in the list. You can simply copy your favorite implementation of your favorite sorting algorithm to get the elements of the supplied list in sorted order.
2. (a) Define a SCHEMEprocedure, named `(fold-right op initial sequence)` which accumulates the values in the list `sequence` using the function/operator `op` and initial value `initial`. `fold-right` should start with the initial value and accumulate the result from the last item in the list to the first. The procedure is named "fold-right" because it combines the first element of the sequence with the result of combining all the elements to the right. For example:

```
(fold-right + 0 (list 1 2 3 4 5))
15
(fold-right * 1 (list 1 2 3 4 5))
120
(fold-right cons '() (list 1 2 3 4 5))
(1 2 3 4 5)
```

   (b) Define a SCHEMEfunction, named `(fold-left op initial sequence)`, which is another accumulate procedure except that `fold-left` applies the operator to the first element of the list first and then the next until it reaches the end of the list. That is, `fold-left` combines elements of `sequence` working in the opposite direction from `fold-right`.

```
(fold-left + 0 (list 1 2 3 4 5))
15
(fold-left * 1 (list 1 2 3 4 5))
120
(fold-left cons '() (list 1 2 3 4 5))
(5 4 3 2 1)
```

   (c) Complete the following definition of my-map below which implements the `map` function on lists using only the `fold-right` function.

```
(define (my-map p sequence)
(fold-right (lambda (x y) <??>) '() sequence))
```

(d) Complete the following definition of my-append below which implements the append function on lists using only the `fold-right` function.

```
(define (my-append seq1 seq2) (fold-right cons <??> <??>))
```

(e) Complete the following definition of my-length below which implements the `length` function on lists using only the `fold-right` function.

```
(define (my-length sequence) (fold-right <??> 0 sequence))
```

(f) Complete the following definition of `reverse-r` below which implements the `reverse` function on lists using only the `fold-right` function.

```
(define (reverse-r sequence)
(fold-right (lambda (x y) <??>) '() sequence))
```

(g) Complete the following definition of `reverse-l` below which implements the `reverse` function on lists using only the `fold-left` function.

```
(define (reverse-l sequence)
(fold-left (lambda (x y) <??>) '() sequence))
```

(h) [SICP EXERCISE 2.34] Evaluating a polynomial in x at a given value of x can be formulated as an accumulation. We evaluate the polynomial

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

using a well-known algorithm called *Horner's rule*, which structures the computation as

$$(\cdots(a_n x + a_{n-1})x + \cdots + a_1)x + a_0$$

In other words, we start with a_n, multiply by x, add a_{n-1}, multiply by x, and so on, until we reach a_0. Use the `fold-right` function to define a SCHEMEprocedure, named `(horner-eval x coefficient-list)` which evaluates a polynomial using Horner's rule. Assume that the coefficients of the polynomial are arranged in a list, from a_0 through a_n.
For example, to compute $1+3x+5x^3+x^5$ at $x=2$ you would evaluate

```
(horner-eval 2 (list 1 3 0 5 0 1))
```

CSE1729 - Introduction to Programming October 28, 2019

Laboratory Assignment 7

Objectives

- Work with trees
- Use Binary Search Trees

Activities

Trees

1. Write a SCHEME function named `tree-depth` which takes a tree node, n, as a parameter and returns the depth of the tree rooted at n. For the purpose of this exercise, the depth of a tree rooted at n is one plus the maximum of the depths of its two children. Recall the following code from the lecture slides:

```
(define (make-tree value left right)
  (list value left right))

(define (value T) (car T))
(define (right T) (caddr T))
(define (left T)  (cadr T))

(define (insert x T)
  (cond ((null? T) (make-tree x '() '()))
        ((eq? x (value T)) T)
        ((< x (value T)) (make-tree (value T)
                                    (insert x (left T))
                                    (right T)))
        ((> x (value T)) (make-tree (value T)
                                    (left T)
                                    (insert x (right T))))))
```

2. Define a SCHEME function named `(count-pred P tree)` which given a binary tree and predicate function, `P`, applies the predicate to each of the values in the tree and returns the number of values for which the predicate returns `\#t` (true).

3. Define a SCHEME function named `count-one-child` which returns the number of internal nodes of a binary tree which have exactly one child.

Binary Search Trees

4. Define a SCHEME function named `(invert-bst T)` which, given a Binary Search Tree (BST) *inverts* that BST. That is, the function returns a binary tree in which has the property that all values at nodes in the left subtree are greater than the value at the root node and all values at nodes in the right subtree are less than the value at the root node.

5. Write a SCHEME function, named `(num-gteq-z z T)` which, given a binary search tree T and an integer z, returns the number of integers in the tree that are greater than or equal to z. For example, given the tree

below and the number 5, your function should return 4, since there are 4 numbers in the tree that are greater than or equal to 5 (specifically, 5, 8, 11, and 21).

Your solution should exploit the fact that the tree is a binary search tree to avoid considering certain subtrees.

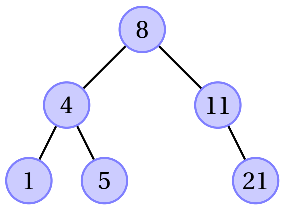


Figure 1: A binary search tree with 6 nodes, 4 of which are 5 or larger.

6. **Organization Trees** Businesses build organizational charts showing the management structure of the business. With some modifications to expand on our binary trees, we can store the same information in a tree structure and perform some computation on that data structure. There will be two modifications necessary. The data stored in each node will consist of two pieces of information, a name and the salary for the person represented by the node. These two pieces of information will be stored together in a pair. The second modification is necessary for managers to be able to manage more than two subordinates. Subordinates will be stored in a list of organization trees which are all subtrees of the node. See Figure 2 for an example in which the top manager has two "direct reports." One direct report of Bob is Mary, who has only one direct report. The other is Steve who has three direct reports. See the convenience functions defined below as well as the SCHEME code for an example organization tree (org-tree).

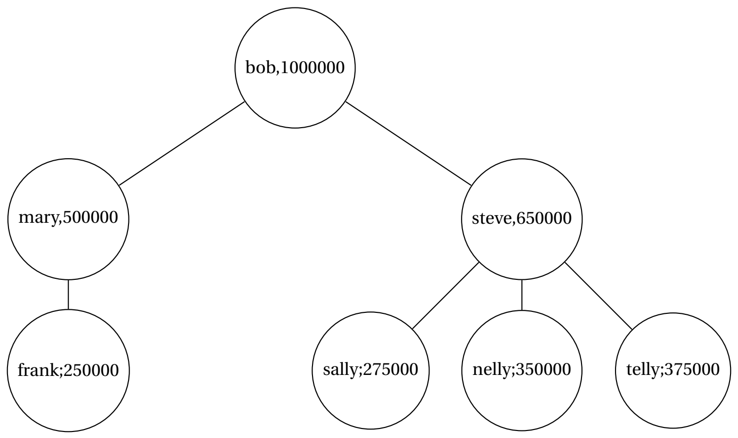


Figure 2: An organizational chart stored in a tree.

```
(define (make-org-tree name salary direct-reports)
  (list (cons name salary) direct-reports))
(define (get-name org-tree) (caar org-tree))
(define (get-salary org-tree) (cdar org-tree))
(define (get-direct-reports org-tree) (cadr org-tree))
(define org
  (make-org-chart
   "bob"
   1000000
(list (make-org-chart "mary"
                      500000
                      (list (make-org-chart "frank" 250000 (list))))
      (make-org-chart "steve"
                      650000
                      (list (make-org-chart "sally" 275000 (list))
                            (make-org-chart "nelly" 350000 (list))
                            (make-org-chart "telly" 375000 (list))))))
```

(a) Define a SCHEME procedure, named `(num-direct-reports org-tree)`, which, given an organizational chart stored in a tree, returns the number of direct reports for the employee at the root of the tree.

(b) Define a SCHEME procedure, named `(dept-size org-tree)`, which, given an organizational chart stored in a tree, returns the number of employees in the department defined by the root of the given tree. Include the manager at the root of the tree plus the number of employees working under that manager (sum of all the direct-reports, their direct reports and so on).

(c) Define a SCHEME procedure, named `(dept-budget org-tree)`, which, given an organizational chart stored in a tree, returns sum of all the salaries in the entire department defined by the root of the given tree. Include the salary of the manager at the root of the tree as well as employees working under the manager (sum of all the salaries of the direct-reports, their direct reports and so on).

CSE1729 - Introduction to Programming November 4, 2019

Laboratory Assignment 8

Objectives

- Work with heaps

Activities

1. For this question, we will use the heap-supporting functions as seen in the lecture slides as building blocks for this assignment to build a new heap implementation. The heap implementation shown in the lecture slides is an example of a min-heap, in which the smallest element is at the root and all elements in child trees are larger than the value at the root. We can also construct max-heap data structures in which the largest element in the heap is at the root and all elements in child trees are smaller than the root.

   The objective is to define SCHEME functions to manipulate a heap which:

   1. maintain a binary tree as a heap,
   2. use a generic (first order) order relation,
   3. provides functions which can determine if a heap is `empty?` as well as `heap-insert`, `heap-remove`, and `combine-heaps`

   (a) Define a SCHEME procedure, named `(heap-insert f x H)`, which adds element `x` to heap `H` using the first-order relation `f` to determine which element belongs at the root of each (sub)tree.
   For instance, if we wanted the same behavior as the heaps in the lecture slides (min-heap), we would use the "less than" function as our first-order relation:

   ```
   (heap-insert < 100 (heap-insert < 10 (list)))
   (10 () (100 () ()))
   ```

   If, instead, we wanted a max-heap implementation, where the largest element is at the root of the heap, we would use the "greater than" function as our first-order relation.

   ```
   (heap-insert > 100 (heap-insert > 10 (list)))
   (100 () (10 () ()))
   ```

   Note, you must use the same first-order relation for all of the heap procedures applied to a particular heap structure.

   (b) Define a SCHEME procedure, named `(combine f Ha Hb)`, which accepts three arguments, `f`, a first-order relation which is used to order the elements in the heap, and two heap structures, `Ha` and `Hb`, which have been constructed using the same first-order relation. For example, for two min-heaps

   ```
   (define Ha (heap-insert-list > (list 9 5 7 3) (list)))
   (define Hb (heap-insert-list > (list 2 8 4 6) (list)))
   (combine > Ha Hb)
   (9 (7 () (5 () (3 () ()))) (8 (4 () ()) (6 () (2 () ()))))
   ```

   (c) Define a SCHEME function, named `(empty? H)` which takes one argument, a heap, and returns a boolean value, true if `H` is the empty heap and false otherwise.

(d) Define a SCHEME function, named (heap-remove f H) which takes two arguments, a heap and a first-order relation, a heap containing the elements of H with the root value removed. Note, heap-remove must take the first-order relation as a parameter to pass to other functions.

```
(heap-remove > (combine > Ha Hb))
(8 (6 (2 () ()) (4 () ())) (7 () (5 () (3 () ()))))
```

CSE 1729 - Introduction to Principles of Programming November 19,2019

Laboratory Assignment 9

Objectives

- Work with mutations
- Work with objects

Activities

1. Early in the course, we saw how to compute the factorial function using nondestructive functional programming. We've seen that a good way to do this is to define helper function which passes itself an accumulator at each recursive call. Using destructive assignment, we can use a helper function with no arguments that can update an accumulator in place rather than passing it as an argument to itself. This helper function doesn't even need to evaluate to anything useful, as the result ends up in the in-place accumulator.

   What does this look like? Here is an example that takes this approach to sum the first n integers:

```
(define (sum-first-n n)
  (let ((sum 0)
        (count 0))
    (define (helper)
      (cond ((= count n) 'done)
            (else
             (set! count (+ count 1))
             (set! sum (+ sum count))
             (helper))))
    (helper)
    sum))
```

   Notably, when `helper` is finished, the result is in the variable `sum`.

   Write a new version of the factorial procedure `(fact n)` that works this way (that is, using a `helper` with no parameters that modifies an accumulated value). Make sure it works as intended by testing it on the numbers 1 through 5.

2. Extend the bank account example in the slides with the following upgrade and new methods:

   - `withdraw` – modify this method so it prints a message showing what the balance is and how much less than the request it is (i.e. "Withdrawal not allowed since balance is x" where x is the current balance of the account). if the request is larger than the balance.
   - `accrue` – add 1 year of simple interest to the balance. At first assume an interest rate of 1%
   - `setrate` – change the interest rate to the given argument, where 1% is represented as 0.01, not 1.

You may start with the following code which is (essentially) copied from the lecture slides:

```scheme
(define (new-account initial-balance)
  (let ((balance initial-balance))
    (define (deposit f)
       (set! balance (+ balance f))
        balance)
    (define (withdraw f)
      (cond ((> f balance)  "Insufficient funds")
            (else
               (set! balance (- balance f))
               balance)))
    (define (bal-inq) balance)
    (lambda (method)
      (cond ((eq? method 'deposit) deposit)
            ((eq? method 'withdraw) withdraw)
            ((eq? method 'balance-inquire) bal-inq)))))
```

3. Create two bank account objects and demonstrate that the balance and rate can be set and modified independently.

4. **(Pseudorandom Number Generator, PRNG)** Randomly chosen numbers are often needed for computer simulations. Different methods have been devised for generating numbers that have properties of randomly chosen numbers. Because numbers generated by systematic methods are not truly random, they are called pseudorandom numbers. The most commonly used procedure for generating pseudorandom numbers is the linear congruential method. We choose four integers: the *modulus* m, *multiplier* a, *increment* c,and *seed* x_0, with $2 \leq a < m$, $0 \leq c < m$, and $0 \leq x_0 < m$. We generate a sequence of pseudorandom numbers $\{x_n\}$, with $0 \leq x_n < m$ for all n, by successively using the recursively defined function

$$x_{n+1} = (ax_n + c) \bmod m$$

Most computers do use linear congruential generators to generate pseudorandom numbers. Often, a linear congruential generator with increment $c = 0$ is used. Such a generator is called a pure multiplicative generator. For example, the pure multiplicative generator with modulus $2^{31} - 1$ and multiplier $7^5 = 16,807$ is widely used. With these values, it can be shown that $2^{31} - 2$ numbers are generated before repetition begins.
Pseudorandom numbers generated by linear congruential generators have long been used for many tasks. Unfortunately, it has been shown that sequences of pseudorandom numbers generated in this way do not share some important statistical properties that true random numbers have. Because of this, it is not advisable to use them for some tasks, such as large simulations. For such sensitive tasks, other methods are used to produce sequences of pseudorandom numbers, either using some sort of algorithm or sampling numbers arising from a random physical phenomenon.
Define a SCHEME object representing a pure multiplicative pseudorandom number generator, using the modulus and multiplier above, that provides the following operations:

next calculates and returns the next pseudorandom number

get retrieves the current pseudorandom number in the sequence

Your object creation function, prng should accept a value for the initial seed as well as the minimum and maximum values desired for pseudorandom numbers. You can restrict the large numbers provided by the linear congruential generator by applying the modulo function to it with the size of the range of numbers required (i.e. $x_n modulo(max - min)$ and shifting it up to the desired range by adding min . Your object should look something like,

```
(define (prng seed min max)
  (let (...)                 ;;internal prng variables
        ...                 ;;prng methods
    (lambda (method)         ;the dispatcher
      (cond ((eq? method 'next) next)
            ((eq? method 'get) get)))))
```

CSE1729 - Introduction to Programming December 2, 2019

Laboratory Assignment 10

Objectives

- Work with streams.

Activities

There are simple ways to check a number for primality if a candidate number is quite small. For instance, you can just see if any numbers between 1 and that candidate divide the candidate evenly. You can use a slightly smarter algorithm that only checks to see if the candidate is divisible by prime numbers less than the candidate, and even smarter, primes less than the square root of the candidate.

The largest known prime number is $2^{82,589,933} - 1$, a number with $24,862,048$ digits. It was found by the Great Internet Mersenne Prime Search (GIMPS) in 2018 (see https://www.mersenne.org/). The majority of the largest known primes are Mersenne Primes, that is, primes of the form $2^p - 1$ where p is an odd prime. The reason most of the largest prime numbers are of this form is because there is an easy (faster) way to verify primality for Mersenne Primes. Note, however, that even with this method, it can take approximately one month to conduct the necessary computations to verify a number is a Mersenne prime. Édouard Lucas was able to verify

$$2^{127} - 1 = 170,141,183,460,469,231,731,687,303,715,884,105,727$$

is prime by performing the following calculations *by hand* in 1876. It would take until 1951 for Aimé Ferrier to find the next largest prime using a mechanical calculator.

1. **Lucas Pseudoprimes**
   (a) One test for primality uses the Lucas sequence, which is similar to the Fibonacci sequence, except it starts with the values 1 and 3. So, it is defined as follows:

$$L_n = \begin{cases} 1 & \text{if } n = 0 \\ 3 & \text{if } n = 1 \\ L_{n-1} + L_{n-2} & \text{otherwise} \end{cases}$$

   Define a SCHEME function, named `(lucas-stream)`, which evaluates to a stream object which produces the stream of Lucas numbers.

   (b) To show how to check primality with the Lucas sequence, let's look at an example. Consider the first few numbers in the Lucas sequence:

$$1, 3, 4, 7, 11, 18, 29, 47, 76, 123, \ldots$$

   If we are interested to see if a particular number, say 7, is prime, we take the 7^{th} number in the Lucas sequence and subtract 1. The 7^{th} Lucas number is 29. Subtract 1 and we have 28. Now, if this result is evenly divisible by 7, 7 is *probably* prime. In our example, 28 is divisible by 7, so it is probably prime. We call these numbers Lucas pseudoprimes.
   If we were interested in another number, say 8, we would take the 8^{th} Lucas number and subtract 1. In this case, we get 46, which is not evenly divisible by 8. So, 8 is *definitely* not prime. This method is useful for filtering out composite numbers.
   So, the general procedure to determine if an integer p could possibly be prime is as follows:

   1. Find the p^{th} Lucas number
   2. Subtract 1

3. Check to see if p divides the result evenly

Define a SCHEME function, named `(lucas-pseudoprime p)` which uses this approach to determine if an integer p is a Lucas pseudoprime.

2. This Lucas pseudoprime solution isn't completely satisfying. We'd like a more definitive solution to test very large numbers for primality. Consider the following solution for verifying Mersenne primes.
   (a) To check to see if a number is a Mersenne prime, we first need to compute values in the Lucas-Lehmer sequence which is defined as follows:

$$LL_i = \begin{cases} 4 & \text{if } i = 0 \\ LL_{i-1}^2 - 2 & \text{otherwise} \end{cases}$$

   Define a SCHEME function, named `(ll-stream)`, that returns a stream object which produces the stream of Lucas-Lehmer numbers. If you use your stream to find the first six numbers or so, you will see that these numbers grow *very* quickly.

$$4, 14, 194, 37634, 1416317954, 2005956546822746114, 4023861667741036022825635656102100994, \ldots$$

   (b) To use this sequence to determine if a number of the form $2^p - 1$ is prime, take the exponent, p, and find the $(p-1)^{st}$ number in the Lucas-Lehmer sequence. If $2^p - 1$ divides this number in the sequence evenly, then $2^p - 1$ is definitely prime. If the candidate number does not divide this number in the sequence, it is definitely not prime. For example, if we wanted to verify $2^3 - 1 = 7$ is prime, we would find the $3 - 1 = 2^{nd}$ number in the Lucas-Lehmer sequence (14), and check to see if 7 divides 14 evenly. 7 does divide 14 evenly. So, 7 is definitely prime.
   As presented, this procedure would require us to work with ridiculously large integers to verify the primality of large Mersenne primes. However, there is hope. Note that, to determine primality, we will need to determine if our candidate number $(2^p - 1)$ divides the corresponding number in this sequence evenly. So, all we really need to know are the remainders of dividing each of the numbers in the Lucas-Lehmer sequence by the candidate number.
   Define a SCHEME function, named `(ll-test-stream c)` which produces a sequence of integers the corresponding number in the Lucas-Lehmer sequence modulo the candidate number c. Your function should not use the `(ll-stream)` function, but should compute the square of the previous term, subtract two, and produce that result modulo p. You should be able to copy your solution to the Lucas-Lehmer sequence stream and add the modulo operation.

$$LL(i,c) = \begin{cases} 4 & \text{if } i = 0 \\ [LL(i-1,c)^2 - 2] \text{ modulo } c & \text{otherwise} \end{cases}$$

   As an example, if we wanted to test the integer $2^{13} - 1 = 8191$ for primality using this method, we would produce the sequence

$$4, 14, 194, 4870, 3953, 5970, 1857, 36, 1294, 3470, 128, 0, \ldots$$

   We can see that the $13 - 1 = 12^{th}$ number in this sequence shows a remainder of 0. So, $2^{13} - 1 = 8191$ is prime.

3. Define a SCHEME function, named `(mprime?  p)` which takes the power, p, of an integer of the form $2^p - 1$ and uses the `(ll-test-stream c)` function to test for primality using the method outlined above. Specifically, if the $(p-1)^{st}$ integer in the corresponding `ll-test-stream` sequence is 0, the number $2^p - 1$ is prime. Note, that $c = 2^p - 1$ when using your `(ll-test-stream c)` function. Also, note that the test stream starts with index $i = 0$.
   Note, Lucas was able to determine $2^{127} - 1$ was prime. He was also able to determine that $2^{67} - 1$ was not prime, *without finding a single factor!*