Exercise 2: Scheme introduction

Deadline exercise 2: Monday Dec. 3rd, 08.00

Task 1: Basic list procedures

a) Write a procedure that reverses a list. Write both a recursive and an iterative process. Hint: Use `Append` to add one list to another.

Procedure:

```
reverse: list -> list
```

Example:

```
(reverse (list 1 2 3 4 5))
;Value: (5 4 3 2 1)
(reverse (list 1 (list 2 3 (list 4)) 5))
;Value: (5 (2 3 (4)) 1)
```

b) Write a procedure that returns the n first elements of a list ($n \leq |\, list \,|$).

Procedure:

```
n-first: number, list -> list
```

Example:

```
(n-first 2 (list 1 2 3 4))
;Value: (1 2)
(n-first 3 (list 1 2 3))
;Value (1 2 3)
```

c) Write a procedure that returns the n last elements of a list ($n \leq |\, list \,|$).

Procedure:

```
n-last: number, list -> list
```

Example:

```
(n-last 2 (list 1 2 3 4))
;Value: (3 4)
(n-first 3 (list 1 2 3))
;Value: (1 2 3)
```

d) Write a procedure that extracts the sub-list from a list specified by the start and end indices (the first element of a list corresponds to index 0).

Procedure:

```
sub-list: number, number, list -> list
```

Example:

```
(sub-list 2 5 (list 1 2 3 4 5 6 7 8 9 0))
;Value: (3 4 5 6)
```

e) Write a procedure `position` that returns the position of the first element in a list that fulfills a certain predicate.

Procedure:
```
position: (list, (element -> boolean)) -> number
```

Example:
```
(position (list 2 4 6 8 9 10 12 13 14) odd?)
;Value: 4
```

f) Modify the previous procedure so that it returns the position of the first element that fulfilled the predicate and that element as a pair.

Procedure:
```
position-and-element:
(list, (element -> boolean)) -> number x element
```

Example:
```
(position-and-element
       (list 2 4 6 8 9 10 12 13 14)
        odd?)
;Value: (4 . 9)
```

g) Once again modify the position procedure to return a list with all position-element pairs that fulfilled the predicate.

Procedure:
```
all-position-and-element:
(list, (element -> boolean)) -> (list of number x
element pairs)
```

Example:
```
(all-position-and-element
       (list 2 4 6 8 9 10 12 13 14)
        odd?)
;Value: ((4 . 9)(7 . 13))
(all-position-and-element
       (list 2 4 6 8 10 12 14)
        odd?)
;Value: ()
```

Task 2: Accumulating lists

Implement the procedures:

- `length`,
- `append`, and
- `map`,

using the `accumulate` procedure (SICP: page 116).

Task 3: Fibonacci numbers

The Fibonacci sequence is one of the top famous sequences in mathematics. It originates from the mathematician by the name Leonardo Fibonacci in the year 1202. The original problem regarded how fast rabbits could breed in ideal circumstances. Suppose that there exist a newly born pair of rabbits, one male and one female, in a field. Rabbits are able to mate at the age of one month, and the pregnancy last for one month. Suppose that the rabbits never die, that they live in monogamy, and that they always mate, i.e. every new month a pair will generate new offspring. Further suppose that the offspring of a pair is always a new pair, one male and one female. The number of pairs of rabbits in the field at the start of each month will then be

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89…,

i.e. the number of rabbits in the field at the start of month *n* (the *nth* Fibonacci number) is defined as:

$$fib(n) = \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ fib(n-1) + fib(n-2) & otherwise \end{cases}$$

From the above definition one can see that after ten years of mating the number of rabbits in the field will be fib(100) = fib(99) + fib(98) = fib(98) + fib(97) + fib(97) + fib(96) = fib(97) + fib(96) + fib(96) + fib(95) + fib(96) + fib(95) + fib(95) + fib(94) = … = 354224848179261915075.

a) Write a procedure that calculates the *nth* Fibonacci number. Write both a procedure that generates a recursive process and a procedure that generates an iterative process.
Experiment by running the two implementations with different *n*'s. Is there any difference in running time? If so, explain why.

b) Write a procedure that generates a list with the first *n* Fibonacci numbers, e.g. requesting the first 10 Fibonacci numbers will return the list '(0 1 1 2 3 5 8 13 21 34) as a result.

Task 4: Tree procedures

a) Write a procedure that squares the leaves of a tree.

Procedure:

```
square-tree: list -> list
```

Example:

```
(square-tree (list 1 2 (list 2 3 4 (list 4 5) 3)
(list 1 2 3)))
;Value: (1 4 (4 9 16 (16 25) 9) (1 4 9))
```

b) Write a procedure that calculates the depth of a tree.

Procedure:

```
depth: list -> number
```

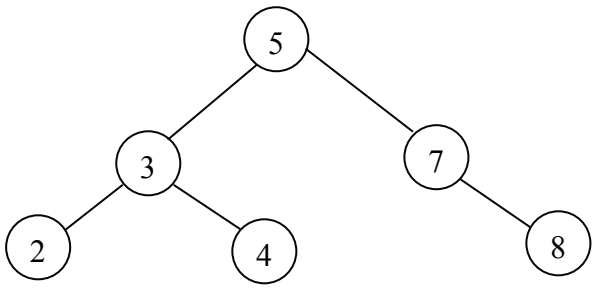
Example:
```
(depth ())
;Value: 0
(depth (list 1 (list 2 3) (list 3 4 (list 5) (list 6
(list 7))) (list 1)))
;Value: 4
```

Task 5: Binary search trees

A binary tree is a tree with at most two children for each node. A binary search tree (BST) is a binary tree where every node's left subtree has values less than the node's value, and every right subtree has values greater than the node's value.

A binary tree can be represented by the recursive structure `(val left right)`. `val` is the value of the root and `left` and `right` are the binary trees for the left and right children or `()` for the empty tree.

The nodes represented by the list `(5 7 3 4 8 2)` can be used to generate the binary search tree:
`(5 (3 (2 () ()) (4 () ())) (7 () (8 () ())))`



a) Write a procedure that constructs a binary search tree from a flat list of unique elements.

Procedure:
```
list-to-bst: list -> list
```

Example:
```
(list-to-bst (list 5 7 3 4 8 2))
;Value: (5 (3 (2 () ()) (4 () ())) (7 () (8 () ())))
```

b) Write the procedures `min-bst` and `max-bst` that finds the minimal and the maximal elements of a BST, respectively.

Procedure:
```
min-bst: list -> number
max-bst: list -> number
```

c) Write the procedure `search-bst` that searches a given BST for a specific value.

Procedure:
`search-bst: list, number -> boolean`