1. [10 points] Describe the three phases of programming language analysis. Use an example of what the input and output of each phase is.

   a. **lexical** – find the tokens, input is the program and the output are the tokens
   b. **syntactic** – group the tokens into sentences defined by some grammar. Input is tokens output is parse trees
   c. **semantic** - process the parse trees by either interpreting them with an interpreter (like Scheme) or compiling them to some lower level machine code, such as C++/Java.

2. [10 points] Give two reasons why programming with pure functions can be beneficial. Explain your answer using a function that is not pure.

   a. You needed to talk about composability and reusability. Use a simple Java function as an example.


   a. What does it mean for a language to be **dynamically typed**? Explain your answer using a concrete example from Scheme that causes a type error.

      i. Types are checked at runtime. In Scheme (+ true “hello”) gives a type error when you try and evaluate the expression.

   b. What does it mean for a language to be **statically typed**? Explain your answer using a concrete example from Java that causes a type error.

      i. Types are checked at compile time. In Java `int a = “fred”;` gives an error when you compile.

4. [10 points] Draw the linked structure for the Scheme list

   \( (\text{list} \ 1 \ (\text{list} \ 3 \ 4) \ 5) \)

   ![Linked Structure Diagram](image)

5. [10 points] Explain the difference between linear and iterative recursion. Address the time and space requirements of each.

   a. A linear recursive function contains one recursive function call with some remaining computation to do after the recursive call returns. This requires stack space that is linear in the number of recursive calls.
b. An *iteratively recursive* function contains one recursive call but there is no computation to do after the recursive call returns. Hence the recursion can be replaced with a loop requiring constant stack space.

6. [10 points] Write a Scheme predicate `forall` that takes a predicate `p` and a list `L` and returns true if every item in `L` satisfies `p`. `forall` should be iteratively recursive. Do not use `map`, `fold`, or `filter` to implement `forall`. Here is an example of how `forall` should be called `(forall even? (list 1 2 3 4 5))`

The trick is that it must be iterative. Here's one solution.

```scheme
(define (forall1 p? l)
  (if (null? l) true
      (if (not (p? (car l))) false (forall1 p? (cdr l)))))
```

7. [10] Write `forall` using `map` and `fold`. The interface should be the same as in question 6.

```scheme
(define (forall2 p? l)
  (foldl (lambda (a b) (and a b)) true (map p? l)))
```

8. [10 points] Write a Scheme function `leaves` that returns the number of leaves in a binary tree, where binary trees are represented like we did in class. You can assume `left`, `right`, `root`, and the predicate `leaf?` are defined for you.

```scheme
(define (leaves t)
  (if (null? t) 0
      (if (leaf? t)
          (+ 1 (leaves (left t)) (leaves (right t)))
          (+ (leaves (left t)) (leaves (right t))))))
```

9. [10 points] Consider a list of numbers such as `(list 1 2 3 4 5)` Write a Scheme expression that computes \( \sum_{i} \left( (x_i - 1)^2 + 1 \right) \) where \( x_i \) refers to the \( i \)th item in the list. Full credit for being concise (e.g., `map`, `fold`, etc.) and is space and time efficient at runtime.

```scheme
(define (q9 l)
  (foldl + 0 (map (compose add1 sqr sub1) l)))
```

10. [10 points] Write a Scheme function `fibs` that takes an integer parameter `n` and returns a list of the first `n` fibonacci numbers in increasing order. For example,

```scheme
(fibs 3) is (list 1 1 2) and (fibs 10) is (list 1 1 2 3 5 8 13 21 34 55)
```
The function \texttt{fibs} must compute the list efficiently.

Here is Susan's solution (which was better than mine).

\begin{verbatim}
(define (fibs n)
  (fibHelper n 1 1 '()))

(define (fibHelper n x y acc)
  (if (> n 0)
      (fibHelper (- n 1) y (+ x y) (cons x acc))
      (reverse acc)))
\end{verbatim}