1. **Classify the following as a syntax error, static semantic (contextual) error, or not a compile time error.**

    a. **x+++-y: Not a compile time error.**

This is valid syntax parsed as (x++) + (-y)

    b. **x---+y: Not a compile time error.**

This is valid syntax parsed as (x--) - (+y)

    c. **Incrementing a read-only variable: Static semantic (contextual) error.**

This is caught at compile time because it violates type rules.

    d. **Code in class C accessing a private field from class D: Static semantic error.**

This is a visibility/access control violation caught at compile time.

    e. **Using an uninitialized variable: Static semantic error.**

Java's compiler checks for definite assignments before use.

    f. **Dereferencing a null reference: Not a compile time error.**

This is a runtime error (NullPointerException).

    g. **null instanceof C: Not a compile time error.**

This is valid syntax and semantics (will evaluate to false at runtime).

    h. **!!x: Not a compile time error.**

This is valid syntax for double-negation of a boolean.

    i. **x > y > z: Syntax error.**

Java doesn't support chained comparison operators like this.

    j. **if (a instanceof Dog d) {...}: Not a compile time error.**

This is a valid pattern matching syntax.

k. **var s = """This is weird""";: Not a compile time error.**

This is a valid text block syntax.

l. **switch = 200;: Syntax error.**

"switch" is a reserved keyword and cannot be used as a variable name.

m. **x = switch (e) {case 1->5; default->8;};: Not a compile time error.**

This is valid syntax for the switch expression.

2. **How do JavaScript and Rust treat the following:**

   let x = 3;

   let x = 3;

In JavaScript, using let twice for the same variable name in the same scope will cause a SyntaxError. The let declaration is designed to prevent variable redeclaration to avoid accidental bugs.

In Rust, this is perfectly valid and is called "variable shadowing." The second declaration creates a new variable that shadows the first one, effectively replacing it.

3. **Describe how the languages Java and Ruby differ in their interpretations of the meaning of the keyword private.**

There are four major differences: In Ruby, subclasses can use the parent's private methods but in Java, they cannot. In Java, instances of the same class can access each other's private members – In Ruby, you cannot call a private method on another instance, even of the same class. Java's privacy is about class scope, while Ruby's is about method receivers. And finally, Java can have private fields while Ruby uses instance variables with the @ prefix, which have their own visibility rules separate from the private keyword.

4. **Some languages do not require the parameters to a function call to be evaluated in any particular order. Is it possible that different evaluation orders can lead to different arguments being passed? If so, give an example to illustrate this point, and if not, prove that no such event could occur.**

Yes. If the parameters are also function calls to some sort of generator, the order they are evaluated may change what values are assigned to each parameter.

Pseudocode example:

```
// pseudo-generator
let counter = 0
function get_value() {
   return counter++
}

// function that evaluates three parameters
function print_values(a, b, c) {
   print (a, b, c)
}

// Values from the pseudo-generator are passed to the print_values function
// The order the params are evaluated determines the order the numbers get printed in
int main() {
   print_values(get_value(), get_value(), get_value())
}
```

5. **Describe in your own words how the Carlos language handles recursive structs. Describe what kinds of restrictions the language definition imposes and why. Describe how the compiler enforces the restrictions. Write well. Use technical vocabulary accurately.**

The Carlos language handles recursive structs by only allowing data types that can break from the recursive declaration pattern. These data types include optionals and lists since they can be empty, allowing the recursion to terminate. A struct of the same type is not a valid property as the recursion would force an infinite loop of that struct. Functions that have that type of struct as input and or output are also allowed since the struct part of those declarations is not stored inline in the struct.

6. **Some languages do not have loops. Write a function, using tail recursion (and no loops) to compute the minimum value of an array or list in Python, C, JavaScript,**

and in either Go, Erlang, or Rust (your choice). Obviously these languages probably already have a min-value-in-array function in a standard library, but the purpose of this exercise is for you to demonstrate your understanding of tail recursion. Your solution must be in the classic functional programming style, that is, it must be stateless. Use parameters, not nonlocal variables, to accumulate values. Assume the array or list contains floating-point values.

Python:

```
def find_min(arr):
    def _find_min_recursive(arr, current_min=None):
        # Base case 1: Empty array
        if not arr:
            if current_min is None:
                raise ValueError("Cannot find minimum of an empty array")
            return current_min

        # Base case 2: First iteration
        if current_min is None:
            return _find_min_recursive(arr[1:], arr[0])

        # Recursive case
        return _find_min_recursive(
            arr[1:],
            min(current_min, arr[0])
        )

    return _find_min_recursive(arr)
```

Rust:

```
fn find_min(arr: &[f64]) -> f64 {
    // Inner function using tail recursion
    fn _find_min_recursive(arr: &[f64], current_min: Option<f64>) -> f64 {
        match (arr, current_min) {
            // Base case 1: Empty array
```

```
        ([], None) => panic!("Cannot find minimum of an empty array"),

        // Base case 2: First iteration
        (_, None) => _find_min_recursive(&arr[1..], Some(arr[0])),

        // Recursive case: Compare current element with accumulated minimum
        (_, Some(min)) => _find_min_recursive(
          &arr[1..],
          Some(min.min(arr[0]))
        )
      }
    }

    _find_min_recursive(arr, None)
}
```

7. **Your friend creates a little JavaScript function to implement a countdown, like so:**

```
function countDownFrom10() {
  let i = 10;
  function update() {
    document.getElementById("t").innerHTML = i;
    if (i-- > 0) setTimeout(update, 1000);
  }
  update();
}
```

Your other friend says “Yikes, you are updating a non-local variable! Here is a better way:”

```
function countDownFromTen() {
  function update(i) {
    document.getElementById("t").innerHTML = i;
    if (i-- > 0) setTimeout(update(i), 1000);
  }
  update(10);
```

}

What does your second friend's function do when called? Why does it fail? Your friend is on the right path though. Fix their code and explain why your fix works.

The second function incorrectly used setTimeout by using the update function as a parameter instead of a function reference. This caused the recursive calls to execute immediately instead of waiting for the timeout delay.

```
function countDownFromTen() {
  function update(i) {
    document.getElementById("t").innerHTML = i;
    if (i-- > 0) {
      // Pass a function reference, not a function call
      setTimeout(() => update(i), 1000);
    }
  }
  update(10);
}
```

I updated the code to pass a function reference to setTimeout, explicitly waiting for the timeout before decrementing.

8. **Find as many linter errors as you can in this Java source code file (C.java):**

```
import java.util.HashMap;

class C {
   static final HashMap<String, Integer> m = new HashMap<String, Integer>();

   static int zero() {
      return 0;
   }

   public C() {
```

```
    }
}
```

Linter errors:

- Class naming: The class name C is too short and non-descriptive, violating naming conventions.
- Unnecessary generic type specification for HashMap - can use diamond operator <> in Java 7+.
- Empty constructor public C() {} is unnecessary and can be removed.
- Unused static HashMap - no clear purpose or use demonstrated.
- Trivial zero() method could be replaced with a direct return 0 or removed entirely.
- No package declaration, which is recommended for better code organization.
- Lack of documentation/comments explaining the purpose of the class and its members.