Assignment 1 Report

Name: Jessica Vu

Organization of Programming Languages

In this assignment, I implemented the Merge Sort algorithm in five different programming languages: Ada, Go, Python, OCaml, and Prolog. The input (array/list) is hardcoded since I think it's more important to get to know the languages than entering the input every time the program is run.

While the overall task wasn't particularly difficult, I encountered different challenges and advantages in each language, which I will discuss below. This report compares and contrasts the programming experience, covering the ease of implementation, challenges, differences in speed, and my personal preferences for each language.

Merge Sort Approach

I follow the instruction from https://rosettacode.org/wiki/Sorting_algorithms/Merge_sort and used the "divide and conquer" sorting method. It works by recursively breaking down a collection of elements into smaller subarrays, sorting those subarrays, and then merging them back together in the correct order. The algorithm consists of a `merge_sort`, and a `merge` function

1. **Sort Function**:

   It is responsible for recursively dividing the list into two halves. If the list contains only one element or is empty, it is already sorted. Otherwise, the function splits the list in half, sorts each half by calling the sort function recursively, and then call the `merge` function

2. **Merge Function**:

   It takes two sorted lists and combines them into a single sorted list. I compare the smallest elements of both lists and appending the smaller element to the result list. Once one of the lists is empty, the remaining elements of the other list are appended to the result.

1. Python (Original Code)

For python I only have 1 file called `merge_sort.py`

Python was by far the easiest to work with for this task. As the language I'm most familiar with, I was able to write the Merge Sort implementation quickly and efficiently. There was no specific challenge in Python for me, as it is the language I use the most.

To run this python program, simply type in the terminal `python merge_sort.py`

2. Ada

Ada was by far the most challenging language in this assignment. The need to write three separate files and go through the building process (instead of simply running the code) was time-consuming. Its syntax is quite verbose and rigid, which made the process slower compared to the other languages. Specifically:

- `mergesort.ads` ⇒ like a header file, the operators are also defined here
- `mergesort.adb` ⇒ the main logic is here
- `mergesort_test.adb` ⇒ use to build the program

I don't like Ada's verbose and complex syntax. While the strong typing and separation of code into multiple files can be useful in larger projects, for a simple algorithm like Merge Sort, it felt overly cumbersome. Ada's strictness in enforcing practices that promote safety can be frustrating for small or experimental projects.

To run this Ada program, build it first by typing `gnatmake mergesort_test.adb`

When build successfully, print out the output by typing `./mergesort_test`

3. Go

I think Go is pretty easy to work with. Writing the Merge Sort in Go was straightforward, and the language's tooling is easy to remember. I think I just need to get myself familiar with the syntax for a bit, as I feel like Go's syntax is pretty much similar to Python.

Interestingly, Go turned out to be the slowest language in this group. I was really surprised as I thought Ada would give me the longest compiling time. I like Go's simplicity, but its performance in this task was disappointing. But overall, I still want to spend more time with this language as I also known it can be used for backend development.

To run this Go program, type `go run merge_sort.go`

4. OCaml

The hardest part of using OCaml was adapting to its syntax, which is filled with symbols and can be unintuitive for someone like me who is not used to functional programming languages.

I found OCaml's syntax overly complex, with an abundance of symbols making the code difficult to read at times. While functional programming can be powerful, it's not a style I'm particularly fond of, and this experience didn't change my mind. However, the speed and conciseness are clear benefits.

To run this OCaml program, type `ocaml merge_sort.ml`

5. Prolog

Prolog is a very different paradigm from the other languages, which required me to think differently about the problem. Understanding how to handle lists in Prolog took some time, and debugging errors was more challenging than in imperative or functional languages.

Prolog's performance was reasonable but not exceptional. It wasn't the slowest, but it didn't match the speed of Ada or OCaml. I ike Prolog's uniqueness and how it forces a different way of thinking. However, the paradigm shift was a bit of a mental hurdle, and debugging in Prolog can be difficult because the declarative logic can obscure where errors occur.

Run this Prolog program with `?- merge_sort`

Comparison of Programming Experience

- **What was easy?**

Python and Go were the easiest languages to write in due to their simplicity and straightforward syntax. Prolog was easier than expected for recursion due to its natural declarative structure.

- **What was hard?**

  Ada was by far the hardest, with its verbose syntax and multi-file structure adding unnecessary complexity. OCaml was also challenging due to its symbolic syntax and functional nature, which required more effort to understand.

- **Noticeable Differences in Speed**

  Go was surprisingly the slowest in this task, while Ada, OCaml, and Python were fairly fast, with Prolog sitting somewhere in between. Given Go's usual reputation for speed, this was unexpected.

- **Likes/Dislikes**

  I liked Python’s ease and readability the most, followed by the clean syntax of Go. I disliked Ada's cumbersome structure and OCaml's overly symbolic syntax. Prolog, while interesting, required a major shift in thinking, which could be both a pro and a con depending on the task.