Chapter 9 Assignment – Languages & Paradigms

Ryan Fanning

Part 1: Subprograms (15 points) (read section 9.2 in textbook)

A subprogram is a block of code designed to perform a specific task within a larger program. It can be called from different parts of the program whenever that task is needed, making the code more organized and easier to maintain. Subprograms usually have a name, may take input parameters, and can return a value. They promote modularity and reusability by allowing programmers to break complex problems into smaller, manageable sections that can be developed and tested independently.

There are two main types of subprograms: procedures and functions. The key difference between them is that a function returns a value, while a procedure does not. A procedure is mainly used to perform an action, such as displaying output or modifying data, whereas a function performs a calculation and returns the result to the caller. For example, in Python, a procedure might print a greeting **(def greet_user(name): print("Hello", name)),** while a function might return a sum **(def, add(a, b): return a + b)**. In this way, functions are often used within expressions, while procedures are called as standalone actions.

Using subprograms in software development offers several advantages. They improve code readability and maintainability by dividing programs into logical sections. They also encourage reusability since the same subprogram can be called multiple times, reducing redundancy. However, subprograms can introduce slight performance overhead due to the extra memory and processing required for function calls. Additionally, managing parameters between subprograms can be complex in large programs. Despite these drawbacks, subprograms remain an essential concept in programming because they make software more structured, reliable, and easier to understand.

Part 2: Parameter passing (15 points) (read section 9.5 in textbook)

Parameter passing is the process of sending data from the main program to a subprogram such as a function or procedure. The main mechanisms for doing this are pass by value, pass by reference, and pass by name. In pass by value, a copy of the actual parameter is passed to the subprogram, meaning any changes made inside the subprogram do not affect the original variable. Pass by reference passes the memory address of the variable, allowing the subprogram to modify the original data directly. Pass by name, used mainly in older languages like Algol, substitutes the argument's expression into the subprogram and evaluates it each time it is used.

For example, in C or Java, parameters are passed by value, so a function like void add(int x) only modifies a copy of x. In C++, parameters can be passed by reference using void add(int &x), which allows changes made in the function to affect the original variable. In contrast, pass by name would behave differently, as the parameter is treated like a textual substitution and evaluated whenever it is used in the subprogram.

The choice of parameter passing mechanism depends on the program's requirements. Pass by value is safe and prevents accidental changes to data, making it ideal for small or constant inputs. Pass by reference is efficient for large data structures and situations where a function needs to modify original data. Pass by name is rarely used in modern programming but can provide flexibility in certain control structures. Understanding these mechanisms helps programmers choose the most appropriate method for efficiency, safety, and clarity in different programming scenarios.

Part 3: Overloaded subprograms (15 points) (read section 9.9 in textbook)

name, as long as they have different parameter lists. This means that the functions must differ in the number of parameters or the type of parameters they take. The main idea is to let programmers use the same function name for similar actions that work with different kinds of data. The compiler decides which version of the function to use based on the arguments given in the function call. This helps make programs easier to read and understand because the same function name can be used for related operations.

For example, in C++, we can overload a function called add so that it can handle both integers and doubles:

```
int add(int a, int b) {
   return a + b;
}

double add(double a, double b) {
   return a + b;
}
```

In this example, both functions are named add, but one works with integers and the other with doubles. When the program calls add(3, 4), the integer version runs, and when it calls add(2.5, 3.5), the double version is used. The compiler automatically picks the correct version based on the type of data passed to the function.

Using overloaded functions has several benefits. It makes code more organized and readable because related functions share the same name. It also saves time since programmers do not need to come up with different names for similar tasks. However, overloaded functions can sometimes cause problems if the compiler cannot clearly decide which version to call, especially when different data types are mixed. Despite this, function overloading is very useful in programming because it helps make code cleaner, easier to read, and more efficient.

Part 4: Generic Functions (15 points) (read section 9.10.1-9.10.4 in your textbook)

Generic functions are special types of functions that can work with different data types without having to write separate versions of the same function. Instead of specifying a specific data type, a generic function uses a type parameter that acts as a placeholder for any data type. This makes the function flexible and reusable because it can perform the same operation on integers, floats, strings, or even user-defined data types. Generic functions are often used in programming languages like C++, Java, and C#, where they help make code more efficient and easier to maintain.

The main difference between generic functions and regular functions is that regular functions are written for specific data types, while generic functions can handle multiple types using type parameters. For example, a regular function might only add two integers or two doubles, requiring separate functions for each data type. A generic function, however, can perform the same operation regardless of the type of data passed to it. This reduces code duplication and makes programs more flexible. Regular functions are simpler to write, but generic functions are more powerful when the same logic needs to apply to different data types.

For example, in C++, a generic function can be written using the template keyword:

```
template <typename T>
T add(T a, T b) {
  return a + b;
}
```

In this example, the type T can represent any data type. When the function is called with integers, T becomes int; when called with doubles, T becomes double. This allows the same function to work with different data types without rewriting it. The main benefit of generic functions is that they make code more reusable and reduce redundancy. However, they can be harder to understand at first, especially for beginners. Overall, generic functions are a powerful feature that helps make programs more flexible, reusable, and easier to manage.

Part 5: Functions as Parameters (15 points) (read sections 9.6 and 9.7 in textbook)

In programming, functions as parameters means passing a function itself as an argument to another function. This allows a function to call or apply another function during its execution. This concept is significant because it enables programmers to write more flexible and reusable code. Instead of hardcoding specific operations, functions can be designed to accept behavior as input, making them adaptable to different tasks without rewriting the same logic multiple times.

For example, in Python, functions can be passed as parameters to other functions:

```
def square(x):
   return x * x

def apply_function(func, value):
   return func(value)

result = apply_function(square, 5)  # result is 25
```

Here, square is passed as a parameter to apply_function. The function apply_function can now apply any function passed to it, not just square. This makes it easy to reuse apply_function with different operations, like doubling or cubing, without creating separate functions for each case.

Using functions as parameters improves code modularity and flexibility. It allows programmers to separate logic from control flow, making programs easier to maintain and extend. For instance, a sorting function can accept different comparison functions, enabling it to sort data in multiple ways without modifying the core sorting algorithm. Overall, this technique helps create cleaner, more general, and highly adaptable programs, which is especially important in large software projects.

Part 6: Coroutine: (10 points) (read section 9.13 in your textbook)

A coroutine is a programming construct that allows a function (or sub-program) to pause its execution part-way through, yield control back to the caller (or scheduler), and then later resume exactly where it left off. Unlike a normal subroutine that runs from start to finish in one go, a coroutine can have multiple “entrance” and “exit” points and preserve its local state between suspensions. Coroutines are significant because they help with cooperative multitasking, asynchronous operations, and managing long-running tasks without blocking the whole program.

For example, in Python you can define a coroutine using the async def and await syntax:

```
import asyncio

async def my_coroutine():
    print("Start")
    await asyncio.sleep(1)
    print("Resumed after 1 second")
    await asyncio.sleep(1)
    print("Finished")

async def main():
    await my_coroutine()

asyncio.run(main())
```

In this example, my_coroutine pauses at each await point, allowing other work (or simply waiting) before resuming. In C# / Unity, a coroutine is often declared with IEnumerator and yield return so it can span multiple frames, e.g.:

```
IEnumerator FadeOut()
{
    for (float alpha = 1f; alpha >= 0; alpha -= 0.1f)
    {
        SetAlpha(alpha);
        yield return null;  // suspend until next frame
    }
}
```

Here, the coroutine retains its loop state across frames.

Coroutines contribute significantly to writing flexible and modular code. Because they can suspend and resume, they allow one function to manage state and progression over time (for example: animations, IO tasks, pipelines) without blocking the rest of the program. This leads to clearer separation of concerns: one coroutine can handle the timing or gradual steps, another can handle user input, and the scheduler just coordinates them. They also reduce the need for complicated callback-chains or thread-management logic, making the code more maintainable. However, developers must ensure they understand when the coroutine is suspended/resumed and the implications for shared state or side-effects.