CS 4337 — Programming Language Paradigms, Homework 1

https://iniyanjoseph.com | iniyanijoseph@gmail.com

What's your take on the idea that "too complex languages are too dangerous to use, so we should keep all languages small and simple"?

I agree with this idea. More complicated languages suffer from several obstacles. They are harder to learn due to their additional syntax, and are less consistently formatted. Although more complex languages may support more features, a well designed simple language can also be extremely powerful, especially in specific applications. This is one of the reasons languages such as Go and Lua have become extremely popular. Their simplicity to use, read and learn have made them popular in industry.

Based on your own experience, is Java a good choice as a first programming language to learn? explain why

Java is a good choice as a first programming language. Although its use in industry has declined from the early 2000s, it has many of the common features of imperative languages. It also gives an introduction to types and pointers without being as difficult as C or C++. It also is easy to run on various machines due to the JVM.

Compare and contrast the imperative and declarative programming paradigms. Provide examples of programming languages that exemplify each paradigm and explain their key characteristics

Imperative languages are designed on the idea of performing operations sequentially. They take in some input, and are told both *what to do and how to do it*. In contrast, declarative languages only require instructions on *what to do, but not how to do it*.
C is one of the most common imperative programming languages in the world. It supports common imperative language features such as variables, looping, and arrays.
In contrast, Haskell is a functional (and thus declarative) programming language. Here, values cannot be modified after creation, functions are first class objects, and only lists (dynamically allocated as linked lists) are supported in the language.

Discuss the importance of language design in the development of programming languages. Identify and explain three design considerations that language designers must consider

The design of a programming language in many ways affects its adoption by users. For example, a language which is extremely difficult to write is unlikely to be adopted by users. Language designers must also consider the reliability of the language for its intended application (i.e. C for cross platform compatibility, Python for ML applications, etc), and its cost of use (the time taken to develop applications with the tool, trains engineers to use it, etc).

Many programming languages distinguish between uppercase and lowercase letters in user-defined names. What are the pros and cons of this design decision?

Distinguishing letters by capitalization can lead to more consistent code. If they are distinguished, then every reference to the variable is guaranteed to be written in exactly the same way. In addition, it leads to more possible short variable names (42^n possible names instead of 26^n) names for a name of length n.
This same requirement of consistency may cause issues compiling code however. Incorrect capitalizations that accidentally create new variables (especially in languages like Python which do not explicitly declare variables before defining them), may lead to incorrect code. Even in languages that explicitly declare variables, code written with arbitrary capitalization schemes can be hard to read due to inconsistency, and can lead to frustrations compiling code.

Explain the meaning of "operator overloading" with examples. Give at least two examples of operator overloading

Operator overloading is when a language allows using reserved operations (often arithmetic operations) for different behaviours based on the types of the operands. For example in python, the addition operator + can be used in multiple ways, such as addition for integers, and concatenation of strings

```
num = 5 + 5
string = "5" + "5"
```

Compare and contrast static and dynamic type systems. Discuss the advantages and disadvantages of each language with programming language examples.

Static typing makes checking for errors at compile time significantly easier. The prototypical example of this is Rust, with its extremely strict compiler. Here, any issues relating to types can easily detected and reported to the user. Thus, code which can compile is often correct.
Dynamic type systems are easier to compile, but may be unsafe or crash during runtime, as they are no longer type safe.

What are the arguments for and against the idea of typeless languages?

Typeless languages can be significantly easier to write. Because they do not have strict rules for types, writing code which can be run can be done easily.
However, this impacts performance very negatively. Because memory cannot be allocated at compile time, every use of memory requires high overhead, which slows the program down. Additionally, they may lead to unsafe behaviour when variables are used unexpectedly.

Describe the importance of memory management in programming languages. Explain the differences between manual and automatic memory management, providing examples of languages that use each approach.

Memory is a limited resource. Therefore, programs which use too much memory have several significant issues. For example memory which is not freed can cause security issues, as the memory can be read after the program is done using it. Additionally, using too much memory can block other programs from running (as the resource is limited).

Programming languages avoid this by freeing memory (i.e. returning it to the operating system). Some languages, such as C, do this explicitly by requiring users to manually request memory with `malloc` and free memory with `free`. In contrast, languages such as Python do this automatically by counting references to objects and periodically removing objects which do not have references. This requires more overhead as the computer must check the memory allocated to each object and reserve additional memory to count references, but also makes the code easier to read and write.
Functional programming languages and Rust are able to free memory with variables in the stack.

What is wrong with the following C++ code and how would you correct it?

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  Int *p = malloc(sizeof(int));
  *p = 42;
  p = malloc(sizeof(int));
  free(p);
}
```

First, "Int" must be lowercased, since the type `Int` is not defined in the program. Secondly, the program leaks memory. The memory claimed by the first `malloc` is never freed from the heap. Thirdly, `malloc` returns a pointer of type `void*`, which must be casted to type `int*` before being assigned to `p`.

To understand the value of recursion in a programming language, write two different C/C++ or Java programs that implement the bubble sort algorithm.

The first program should use a recursive version of the bubble sort algorithm.

The second program should use the non-recursive version of the bubble sort algorithm

```
import java.util.*;
public class BubbleSort {
  public static void main(String[] args){
    int[] li = {1, 2, 5, 3, 8, 7};
    iterativeBubbleSort(li);
    recursiveBubbleSort(li);
    System.out.println(Arrays.toString(li));
  }

  public static void iterativeBubbleSort(int[] list){
    for (int i = 0; i < list.length; i++){
      for (int j = i; j < list.length-1; j++){
        if (list[j] > list[j+1]){
          int temp = list[j+1];
          list[j+1] = list[j];
          list[j] = temp;
        }
      }
    }
  }
```

```
  public static void recursiveBubbleSort(int[] list){
    recursiveBubbleSort(list, list.length);
  }

  public static void recursiveBubbleSort(int[] list, int end){
    if (end <= 0){return;}

    for (int j = 0; j < end-1; j++){
      if (list[j] > list[j+1]){
        int temp = list[j+1];
        list[j+1] = list[j];
        list[j] = temp;
      }
    }

    recursiveBubbleSort(list, end-1);
  }
}
```

Choose a programming language and explore its programming paradigm(s). Discuss the strengths and weaknesses of the chosen language's paradigm(s) and provide examples to support your analysis.

For my language, I chose Rust. The language is extremely capable of safe memory management (hence its popularity in recent years). However, this paradigm of "borrowing" can require additional effort from the programmer.
For example, the following code is incorrect. The programmer must be aware of where a variable will be used in the stack.

```
fn do_something(st){
  println("{}", st);
}
fn main(){
  let x = "Hello";
  do_something(x);
  do_something(x);
}
```

Instead, the programmer must "borrow" the object (much like a reference in C/C++), which blocks `x` from being used in `main`, Prevents `&x` from being modified inside of `do_something`, and makes sure that `x`.

```
struct Point {
    x : i32,
    y : i32,
}

fn do_something(pt:&Point){
  println!("{} {}", pt.x, pt.y);
}
fn main(){
  let x = Point{x:1, y:2};
  do_something(&x);
  do_something(&x);
}
```

There are more nuances than this, but this example basically illustrates the focus of the language. Although it is more difficult to use, it is also proven formally to be memory safe and correct (if the logic is correct).