Lecture 3: Programming Languages Paradigms

because of the difference between each and every other programming language, and the different approaches they take, programming languages took different paths in how they operate and how they are used.
Here we will be seeing the differences in how programming languages operate and dictate the flow of their work.

Compiled vs Interpreted:

Every program is a set of human readable instructions, whether it's to add two numbers or send a request over the internet. Compilers and interpreters take human-readable code and convert it to computer-readable machine code.
But what does that actually mean?
Let's take an example. Imagine there is an ancient Greek recipe, and you, a non Greek speaker, want to know how to do the recipe. Here we will have two ways to do so,we either get a translator and make him translate the text to our language completely, and then follow it, or make him translate the recipe line by line and follow it as he does so. In both cases, we will end up doing the recipe, but the approaches differ, and they carry with them other circumstances. For the first approaches, we will have the recipe with us at all time from there on, and we will be able to do it anytime without the need for our translator, but, in case someone else who doesn't speak our language wanted to do it, we will need to make the translator do the work again and give him a new version in his language.
In the second approach, we won't worry about the language we are translating the recipe to since the translator is with us at all times, but, we will be doing the translation process any time we want to do the recipe.

Same thing when we talk about compiled and interpreted languages. In compiled languages, we compile the code into the machine language we want to run it on, and then run it, but, in case we wanted to run the code on a different machine with a different system, we will need to compile the code all over again. And in case we are working with an interpreted language, we wouldn't worry about the system we are running the code on since we will be translating it into machine code at run time every time we run it.

From this we can conclude the following:

- compiled languages are faster than interpreted languages since they get translated to machine language once and then directly ran run at any given time.
- Interpreted languages are more versatile since they are not system dependent and can run anywhere as long as we have an interpreter.

- Error checking in interpreted languages is easier since the execution is going line by line, hence the error will be detected as it occurs.
- Compiled languages are more efficient in using the system's resources.

A take somewhere in the middle of these two approach was taken when the hybrid form of languages appeared with the Java language, where if has its own compiler that transforms the code into a form closer to the the machine language rather than keeping the language in its original form, and translates it to machine code at run time, which will give us a result that takes from both types advantages while eliminating some of their weaknesses.

Static vs dynamic languages:

Programming languages are often categorized based on how they handle data types and when type checking occurs during the execution of a program. Two common types of languages in this context are **static** and **dynamic** languages.

Static Languages

In static languages, the data types of variables are checked at compile-time. This means that the type of every variable must be explicitly declared or inferred before the code is compiled. Once declared, a variable's type cannot change throughout its lifetime.

Key Features:

1. Type Safety: Errors related to type mismatches are caught early during compilation.
2. Performance: Static languages generally execute faster because type checking is done at compile time, allowing the compiler to optimize the code.
3. Readability: Knowing variable types makes the code easier to understand and maintain.
4. Less Flexible: You cannot assign a different type to a variable without changing its declaration.

Here is an example on how static type declaration happens:

```c
#include <stdio.h>

int main() {
    int num = 10; // Declaration of an integer variable
    float pi = 3.14; // Declaration of a float variable


    printf("Number: %d, Pi: %.2f\n", num, pi);
    return 0;
}
```

As we see, the two variables where explicitly assigned a to a data type and can never be used throughout the rest of the program with other types of data, meaning that the variable named “num” can’t be given a 6.8 value since this value isn’t an integer, and the “pi” variable can’t be given a string value since it won’t be a float.

So trying to do the following

```c
#include <stdio.h>

int main() {
    int num = 10; // Declaration of an integer variable
    float pi = 3.14; // Declaration of a float variable


    printf("Number: %d, Pi: %.2f\n", num, pi);

    num = "Ali"; // this will give a Type-mismatch error when trying to compile this code

    return 0;
}
```

will lead to the following error when we compile the code:

```
ali@AHA:~$ gcc ~/course/Lecture3/Lecture3.c
/home/ali/course/Lecture3/Lecture3.c: In function ‘main’:
/home/ali/course/Lecture3/Lecture3.c:11:9: warning: assignment to ‘int’ from ‘char *’ makes integer from pointer without a cast [-Wint-conversion]
   11 |     num = "Ali"; // this will give a Type-mismatch error when trying to compile this code
      |         ^
ali@AHA:~$
```

Dynamic Languages

In dynamic languages, data types are checked at runtime. Variables do not need explicit type declarations; their types are determined based on the value assigned to them. This allows more flexibility but can lead to runtime errors if type mismatches occur.

Key Features:

1. Flexibility: Variables can store values of different types at different points in time.
2. Ease of Use: Simplifies programming by eliminating the need for type declarations.
3. Slower Performance: Since type checking occurs at runtime, dynamic languages are generally slower than static ones.
4. Potential Errors: Type-related bugs may only appear during execution.

Here is an example on this:

```
num = 10  # Initially, `num` is an integer
print(f"Number: {num}")

num = "hello"  # Now, `num` becomes a string
print(f"String: {num}")
```

Unlike what happened in the previous example where assigning a data of a different type crashed the code, here the type assignment happens dynamically when the code is being interpreted, which will prevent any crashes.
The choice between static and dynamic languages depends on the project requirements. If your focus is on performance, scalability, and early error detection, static languages like C are preferable. On the other hand, if flexibility, rapid development, and ease of learning are priorities, dynamic languages like Python shine.

Strongly typed vs weakly typed languages:

Programming languages can also be classified based on how strictly they enforce type rules. This classification results in strongly typed and weakly typed languages.

Strongly Typed Languages

A strongly typed language strictly enforces type rules and does not allow implicit type conversions (also known as type coercion) unless explicitly handled by the programmer. Any attempt to use a value of one type as another type without explicit conversion will result in an error.

Key Features:

1. Strict Type Enforcement: Types must match exactly or be explicitly converted.
2. Type Safety: Reduces unintended behavior due to implicit type conversions.
3. Error Detection: Prevents subtle bugs that might occur due to type-related assumptions.
4. Clarity: Code is more predictable because types behave as defined.

For example:

```python
num = 10  # Integer
text = "20"  # String

# This will raise a TypeError because Python doesn't allow implicit conversion
result = num + text  # Error: cannot add an integer and a string

# Explicit conversion is required
result = num + int(text)  # Works: 10 + 20 = 30
print(result)
```

This will be considered false since Python is a strongly typed language, so it won't do any under the hood conversion to the values it is passed, and special type conversions are needed.

Weakly Typed Languages

A weakly typed language is more permissive and allows implicit type conversions. This flexibility can lead to unexpected behavior when operations are performed on variables of mismatched types.

Key Features:

1. Type Flexibility: Types can be implicitly converted when needed.
2. Convenience: Easier to work with simple scripts or tasks that don't require strict type safety.
3. Error-Prone: Implicit conversions can cause hard-to-detect bugs.
4. Less Verbose: Reduces boilerplate code since explicit type conversion is often unnecessary.

An example on this would something like:

```c
    int num = 10;
    char A = "A";

    // Here the A will be converted to its ASCII form
    // which is equal to 65, and then it will be treated as an interger
    printf("Result: %d\n", num + A);


    return 0;
}
```

But, there could be some strange behaviors to this, for example:

```c
    int num = 10;
    char A = "A";
    char text[] = "20";

    // Here the A will be converted to its ASCII form,
    // which is equal to 65, and then it will be treated as an interger
    printf("Result: %d\n", num + A);

    // Implicit conversion happens here: string "20" is interpreted as its pointer,
    // since there is no currosponding value to it as what we saw in the previous example
    printf("Result: %d\n", num + text);  // Undefined behavior

    // Explicit conversion fixes the issue
    int converted = atoi(text);  // Convert string to integer
    printf("Result: %d\n", num + converted);  // Output: 30

    return 0;
}
```

So not all cases here will give us a useful result, which will need some extra attention from the user when working with such languages.

Best wishes, Eng. Ali Hassan