Chi-Hao Tu & Yaoyao Liu
COMP 141: Language Design Criteria
Instructions: In this exercise, we are going to review different PL design criteria.

1. Commenting in C

There are two mechanisms in C to declare comments:

1. Commenting in a single line using // at the beginning, and
2. Commenting in potentially multiple lines using /* to mark the beginning and */ to mark the ending.

For each of the following criteria, argue which commenting mechanism is more preferred.

- readability,
- writability, and
- reliability

For **readability,** multi-line comments might be slightly preferred for longer explanations, but single-line comments can enhance clarity for brief annotations.

In terms of **writability,** single-line comments offer ease for short notes, whereas multi-line comments are preferable for longer descriptions or bulk commenting.

Regarding **reliability,** single-line comments may be considered safer due to the lower risk of accidentally commenting out unintended sections of code.

2. Explicitly-typed vs. implicitly-typed PLs

In explicit-typed PLs, the programmer is supposed to annotate each data container with some type. This is while, in implicitly-typed PLs there is not such restriction for the programmer. Compare how this language feature affects the following criteria:

- syntax conciseness,
- maintainability, and
- expressiveness

Syntax Conciseness

Explicitly-typed PLs: These require developers to specify the type of every variable and data structure. This additional requirement can make the syntax more verbose, as each declaration includes both the type and the identifier. While this verbosity can clarify the intended use of each variable, it does reduce syntax conciseness.
Implicitly-typed PLs: In languages that support implicit typing, the compiler or interpreter infers the type of a variable based on the context, such as the value assigned to it. This inference allows for shorter, more concise code because the programmer doesn't need to explicitly state the type. This conciseness is especially noticeable in cases where the type names are long or complex.

Maintainability

Explicitly-typed PLs: The requirement to specify types explicitly can enhance maintainability by making the code more self-documenting. The presence of explicit types can clarify the intended use of variables and function signatures, making the code easier to understand and modify for someone who is not familiar with it. It can also prevent certain types of errors, as the compiler can catch type mismatches at compile time.
Implicitly-typed PLs: While the reduced verbosity of implicitly-typed languages can make code initially quicker to write and potentially easier to read, it may compromise maintainability in the long term, especially in complex codebases. Without explicit type declarations, developers may need to spend more time understanding the intended types of variables or the return types of functions. However, modern development tools and IDEs can mitigate this issue by providing type information on-demand.

Expressiveness

Explicitly-typed PLs: The explicit declaration of types can make the language less expressive in the sense that more code is required to perform simple tasks, especially when dealing with complex types. However, it can also make the language more expressive by clearly communicating the data structures and their intended use, which can be particularly valuable in complex domains where precision is crucial.
Implicitly-typed PLs: Implicit typing can increase expressiveness by allowing developers to write code more succinctly, focusing on what they want to achieve rather than on the details of type management. This can make code more fluid and closer to natural language, potentially speeding up development, especially for prototyping and in domains where flexibility is more important than strict type safety.

3. Semantic safety in C++ vs. Java

1. Consider the following program in C++. Run it and report the result.

```
int main() { int arr[3] = {1,2,3}; cout<<arr[2]<<endl; cout<<arr[4]<<endl; return 0;
}
```

- The program correctly prints the value at index 2 of the array, which is 3.
- Accessing the array at index 4, which is out of bounds, leads to undefined behavior in C++. outcome for this undefined behavior could be any value, such as -3241 in this case.

2. Consider the same program in Java. Run it and report the result.

```
class Main { public static void main(String args[])
        { int[] arr = new int[]{1,2,3};
                System.out.println(arr[2]);
                System.out.println(arr[4]);
        }
}
```

- The program correctly prints the value at index 2 of the array, which is 3.
- When attempting to access the array at index 4, which is out of bounds, Java throws an ArrayIndexOutOfBoundsException. This simulated behavior closely mirrors what would happen in a real Java environment, where the program would terminate with this exception after attempting to access an out-of-bounds index.

3. What can you infer about the semantic safety of C++ vs. Java?

C++ Semantic Safety:

- Less Restrictive at Compile Time
- Undefined Behavior
- Manual Memory Management

Java Semantic Safety:

- Strict Runtime Checks
- Automatic Memory Management
- Exception Handling

In conclusion, Java is considered to be semantically safe. That is, these languages prevent a programmer from compiling or executing any statements or expressions that violate the definition of the language. By contrast, languages such as C and C++ are not semantically safe.

4. Extensibility

Search through the web and find how frequent the following languages get updated.

1. Python
   The first appeared on 20 February 1991 and the latest version on 7 December 2023. Python typically receives a major update once every 12 months, with additional bug-fix updates and security patches released every few months.
2. Java
   The first appeared on May 23 1995 and the latest version on September 19, 2023. Java now follows a faster release schedule, with updates every six months and a Long-Term Support (LTS) version every three years. The fast channel allows early adoption of new features, while LTS versions offer stability for enterprises. Next, I'll search for the update frequencies of Haskell (GHC), Standard ML of New Jersey (SML/NJ), Clojure, and Lua.
3. Haskell (GHC) 1990; 34 years ago
   The first appeared in 1990 and the latest version in July 2010. The Glasgow Haskell Compiler (GHC) is considering a "tick-tock" release cycle to provide clearer guidance for users on migrating between versions and to facilitate planning for GHC releases.
4. ML (look for Standard ML of New Jersey: SML/NJ)
   The first appeared in 1973 and Standard ML of New Jersey (SML/NJ) has adopted a version numbering scheme that reflects the year and the release number within that year, indicated as `YYYY.NN`. This suggests that releases may be planned annually, with potentially multiple releases within a year, but the exact frequency can vary based on development needs and updates.
5. Clojur
   Clojure's stable release version 1.11.1 was made available on April 5, 2022. Additionally, there was a development release, version 1.12.0-alpha5, on October 20, 2023. This information suggests that Clojure has a development cycle that includes both stable and alpha releases, indicating ongoing development and updates to the language. However, the exact frequency of these updates can vary based on development needs and project milestones.
6. Lua
   The first appeared in 1993 and the latest version on 14 May 2023. The frequency of Lua updates can vary depending on various factors such as the introduction of new features, bug fixes, security patches, and community contributions. Lua follows a relatively stable release cycle. Major versions of Lua are not released frequently, and significant updates may take longer intervals, often spanning several years.