Programming Language Concepts

Principles of Programming Languages

Colorado School of Mines

https://lambda.mines.edu
With your learning group:

1. Share your code snippets from the assignment. Explain why one language is **inherently** less maintainable, readable, or abstractable in one language than the other for that particular example.

2. Collectively, as a group, either:
   1. create a **great** definition for **expressivity**
   2. or, create a **great** explanation for how **expressiveness** differs from (and is similar to) **conciseness**

---

**Prove Them Wrong?**

Think that there’s a better way to express the problem for a piece of code your group member is showing? Show an example.

**Remember to be nice.**
Language Implementation Techniques
Compiled Languages

Advantages:
- Runtime is *fast!*

Disadvantages:
- Compile time is slow
- Source code cannot be a part of the input data

Examples

C, C++, and FORTRAN are generally implemented as compiled languages
Interpreted Languages

Advantages:
- No need to compile
- Source code can be a part of input data: you can transmit functions across the network to be run!

Disadvantages:
- Runtime is slow

Examples

BASIC, PHP, and Perl are generally implemented as interpreted languages
Hybrid Interpreters

To speed up the execution of interpreted languages, implementers started getting clever:

- **Interpreted VM Bytecode:** Input is lexed, parsed, then translated to bytecode. The bytecode gets optimized, then the low level bytecode is interpreted. Examples: Python, Java, Ruby

- **Just In Time Compiler:** Source code is compiled as it’s executed, putting machine code on the processor "just in time". Examples: PyPy, LuaJIT, Chrome V8
Hybrid Interpreters

To speed up the execution of interpreted languages, implementers started getting clever:

- **Interpreted VM Bytecode**: Input is lexed, parsed, then translated to bytecode. The bytecode gets optimized, then the low level bytecode is interpreted. Examples: Python, Java, Ruby

- **Just In Time Compiler**: Source code is compiled as it’s executed, putting machine code on the processor "just in time". Examples: PyPy, LuaJIT, Chrome V8

Advantages include all the benefits of interpreted languages, with run times occasionally approaching compiled languages.
Evaluating a Programming Language
Evaluation Metrics

Evaluating programming languages based on:

- **Writability**: How easy is it to write good code?
- **Readability**: How easy is it to read well written code? Is the language easy enough to learn?
- **Reliability**: What features does the language provide to make sure our code works as it is supposed to?
- **Feasibility**: Does an interpreter or compiler actually exist for the platform we need to use? Is it fast enough for our application?
Evaluation Metrics

Evaluating programming languages based on:

**Writability:** How easy is it to write good code?

**Readability:** How easy is it to read well written code? Is the language easy enough to learn?

**Reliability:** What features does the language provide to make sure our code works as it is supposed to?

**Feasibility:** Does an interpreter or compiler actually exist for the platform we need to use? Is it fast enough for our application?

A System of Trade-Offs

Often times, adding features which improve one metric can harm another metric. Examples to come...
The overall simplicity of a language plays a large role in both writability and readability:

- **Feature Multiplicity:** 👍 Writability, 👎 Readability
- **Operator Overloading:** 👍 Writability, 👎 Readability
- **Large Grammars:** 👍 Writability, 👎 Readability

Simplicity can be carried too far. Assembly languages and esoteric languages generally aren’t considered very writable or readable.
The overall simplicity of a language plays a large role in both writability and readability:

- **Feature Multiplicity:** 👍 Writability, 👎 Readability
- **Operator Overloading:** 👍 Writability, 👎 Readability
- **Large Grammars:** 👍 Writability, 👎 Readability

Simplicity can be carried too far

Assembly languages and esoteric languages generally aren’t considered very writable or readable.
Orthogonality: how consistent is the language with itself?

Example of a lack of orthogonality (C++)
Parameters are passed by value, unless they were specified with an &.
Or unless they were an array.

Example of a lack of orthogonality (C/C++)
Arrays can contain data of any type, including pointers.
Unless it's a function pointer.
But you can wrap that function pointer in a struct and you should be fine.

Impacts of poor orthogonality:
poor readability, poor writability, and potentially reduced reliability.
Orthogonality: how consistent is the language with itself?

Example of a lack of orthogonality (C++)

Parameters are passed by value, unless they were specified with an &.
Or unless they were an array.

Impacts of poor orthogonality:
poor readability, poor writability, and potentially reduced reliability.
Orthogonality: how consistent is the language with itself?

Example of a lack of orthogonality (C++)
Parameters are passed by value, unless they were specified with an &. Or unless they were an array.

Example of a lack of orthogonality (C/C++)
Arrays can contain data of any type, including pointers. Unless it’s a function pointer. But you can wrap that function pointer in a struct and you should be fine.

Impacts of poor orthogonality: poor readability, poor writability, and potentially reduced reliability.
**Abstraction**

*Abstraction:* The ability to define and use complicated structures and operations in a way that allows implementation to be ignored.

**Examples:**

- **Functions:** Simplest form of abstraction. Often taken for granted, but gives us easy recursion.
- **Heap Memory:** Imagine trying to create a large unbalanced binary tree in a single-dimensional array.
- **Generics:** Allows us to define operations that apply to multiple data types without reimplementing for each type.
- **Garbage Collection:** A form of automatic memory management.
**Abstraction**: The ability to define and use complicated structures and operations in a way that allows implementation to be ignored.

**Examples:**

- **Functions**: Simplest form of abstraction. Often taken for granted, but gives us easy recursion.
- **Heap Memory**: Imagine trying to create a large unbalanced binary tree in a single-dimensional array.
- **Generics**: Allows us to define operations that apply to multiple data types without reimplementing for each type.
- **Garbage Collection**: A form of automatic memory management.

**With your learning group...**

What other kinds of PL-level abstractions can you name?
Abstraction

**Abstraction:** The ability to define and use complicated structures and operations in a way that allows implementation to be ignored.

**Examples:**

- **Functions:** Simplest form of abstraction. Often taken for granted, but gives us easy recursion.
- **Heap Memory:** Imagine trying to create a large unbalanced binary tree in a single-dimensional array.
- **Generics:** Allows us to define operations that apply to multiple data types without reimplementing for each type.
- **Garbage Collection:** A form of automatic memory management.

With your learning group...

What other kinds of PL-level abstractions can you name?

**Good Abstractions:** 👍 Writability, 👍 Readability, 👍 Reliability
Some languages come with features designed for reliability:

- **Type Checking**: Making sure the type of data can be used with the function or operation you are calling. Independent of static/dynamic: more on this later.

- **Exception Handling**: The ability of a running program to intercept run-time errors and take corrective measures.

- **Taint Protection**: Protects the security of an application by not allowing privileged operations to be performed on tainted data (e.g., user input from a web application).
Reliability Features

Some languages come with features designed for reliability:

- **Type Checking**: Making sure the type of data can be used with the function or operation you are calling. Independent of static/dynamic: more on this later.

- **Exception Handling**: The ability of a running program to intercept run-time errors and take corrective measures.

- **Taint Protection**: Protects the security of an application by not allowing privileged operations to be performed on tainted data (e.g., user input from a web application).

Some features can harm a language’s reliability:

- **Goto**: the ability to jump to different locations in the code without restriction.

- **Aliasing**: allows two different symbolic names (variables, function names, etc.) to refer to the same data. Think pointers in C/C++.
Expressivity

With your learning group:

1. If one language is less expressive than another, how might it be less **writable**?
2. If one language is less expressive than another, how might it be less **readable**?
3. If one language is less expressive than another, how might it be less **reliable**?

Be prepared to share your answers with the class.
Typing Systems
A **binding** refers to the association between:

- a variable and its type,
- a function and its definition,
- a type and its representation (e.g., int is 32-bits),
- or an operation and its symbol (e.g., multiplication is usually *)
A **binding** refers to the association between:

- a variable and its type,
- a function and its definition,
- a type and its representation (e.g., `int` is 32-bits),
- or an operation and its symbol (e.g., multiplication is usually `*`)

**Binding time** refers to the time at which a binding takes place.

**Common Binding Times**

Design time, implementation time, compile time, link time, run time
In a **static typing** system, the binding of a variable to its type occurs *before* run time.

In other words, the type of data is associated with the variable.

```c
int x = 12;
```
In a **static typing** system, the binding of a variable to its type occurs *before* run time.

In other words, the type of data is associated with the variable.

```plaintext
int x = 12;
```

**Advantages:**

- No need to do type checking at run time, this can be done at compile time.
- 👍 Reliability

**Disadvantages:**

- Generics are needed to create operations and functions that apply to multiple types
- 👎 Writability
Dynamic Typing

In a **dynamic typing** system, the binding of a variable to a type occurs *during* run time.

In other words, the type of data is associated with the data itself.

\[ x = \text{int}(12) \]
Dynamic Typing

In a **dynamic typing** system, the binding of a variable to a type occurs *during* run time.

In other words, the type of data is associated with the data itself.

\[ x = \text{int}(12) \]

**Advantages:**

- Collections can be of mixed type without generics, functions can take multiple types without generics
- Types can be dynamically created at run time
- 👍 Writability

**Disadvantages:**

- Type checking must be done at run time; makes things slow
- ⚠️ Reliability
Untyped Systems

In an **untyped** system, variables are never bound to a type. In other words, the functions and operations called on the variables determine the type:

```plaintext
12 x define
  x int->string print-string
```

**Note**

Don’t confuse untyped for type inference. Type inference is generally used with static typing systems.
Untyped Systems

In an **untyped** system, variables are never bound to a type. In other words, the functions and operations called on the variables determine the type:

\[
12 \ x \ \text{define} \\
x \ \text{int->string} \ \text{print-string}
\]

**Note**

Don’t confuse untyped for type inference. Type inference is generally used with static typing systems.

**Advantages:**

- No need to do type checking, ever.
- 👍 Feasibility

**Disadvantages:**

- 👎 👎 👎 Reliability
Strong and Weakly Typed

- **Type safety** means a language will not allow bits to be interpreted as the incorrect data type. For example: treating the bits of an integer as a floating point number.

- **Implicit type conversions** are when a language will automatically convert data types to allow an expression to be computed.

- **Strongly typed** programming languages are both type safe and do not allow implicit type conversions.

- **Weakly typed** programming languages are either not type safe or allow implicit type conversions.

- Whether a language is strongly or weakly typed has **nothing** to do with whether it is statically/dynamically typed, or compiled/interpreted.
## Type Systems: Language Examples

<table>
<thead>
<tr>
<th></th>
<th>Strong</th>
<th>Weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Java, Haskell, Rust, Go</td>
<td>C, C++</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Python, Ruby</td>
<td>PHP, JavaScript</td>
</tr>
</tbody>
</table>
End of Lecture: Roadmap
Before you leave class, divide up LGA-02 with your group members.

I’m going to be in Germany 1/17 thru 1/23. R. Blake Jackson will lecture during this time.

No office hours on Monday 1/22, as I won’t be in the country.

Your first quiz will be Thursday, 1/25. Don’t expect it to be too hard.

Your first programming assignment will be due Friday, 1/26 at midnight. I will post the assignment later today.