PL Evaluation and Typing Systems

Principles of Programming Languages
Colorado School of Mines

https://lambda.mines.edu
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.

For example, these features are non-simple:

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

The overall simplicity of a language plays a large role in both writability and readability.

For example, these features are non-simple:

- **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?
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?

<table>
<thead>
<tr>
<th>Example of a lack of orthogonality (C++)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters are passed by value, unless they were specified with an &amp;.</td>
</tr>
<tr>
<td><em>Or unless they were an array.</em></td>
</tr>
</tbody>
</table>

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

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

With your learning group...

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

With your learning group...

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

**Good Abstractions:** 👍 Writability, 👍 Readability, 👍 Reliability
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).
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

**Expressivity:** How easy is it for the programmer to express their solution to a problem in the language?

With your learning group:

1. Think of a scenario and two programming languages, where expressing a solution to the problem might be easier in one language than another.

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

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

4. If one language is less expressive than another, how might it be less **reliable**?

Be prepared to share your answers with the class.
Type Systems
When we refer to "type systems", we aren’t just talking about how you have to write the type of a variable in C, whereas you don’t in Python. There’s a lot that goes in:

- Static/Dynamic Typing
- Untyped Systems
- Implicit/Explicit Type Specification
- Strong/Weak Typing
- Gradual Typing
- Duck Typing -- covered later in the course
Explicit/Implicit

In a language with **explicit type specification**, the type of a variable *must* be specified:

```plaintext
int x = 10;
```

In a language with **implicit type specification**, the type of a variable need not be specified:

```plaintext
var x = 10
```

**Note**

Explicit/implicit has nothing to do with static/dynamic. We will talk about that in a second...
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 include:

- Design time
- Implementation time
- Compile time
- Link time
- Run time
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 **statically typed** language, the language design makes it possible to bind the type of any piece of data **before run time**.
Static Typing

In a **statically typed** language, the language design makes it possible to bind the type of any piece of data **before run time**.

**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 **dynamically typed** language, the language design makes it possible to bind the type of any piece of data **during run time**. Commonly, the type of data is associated with the data itself.
Dynamic Typing

In a **dynamically typed** language, the language design makes it possible to bind the type of any piece of data **during run time**. Commonly, the type of data is associated with the data itself.

**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

**Gradual Typing**

**Gradual Typing** can be used to refer to a language which allows optional explicit typing in a dynamically typed language.
In an **untyped** language, data cannot be bound to a type.

Commonly, the functions and operations called on the variables determine the type.
Untyped Systems

In an **untyped** language, data cannot be bound to a type.

Commonly, the functions and operations called on the variables determine the type.

**Advantages:**

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

**Disadvantages:**

- Reliability
Strongly 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.

**Note**

By definition, untyped languages are also weakly typed.
Given the code snippet from a fake language below:

```plaintext
int a = 10
a += 5
print(a)
```

- Explicit or implicit?
- Is it possible that the language is statically typed?
- Is it possible that the language is dynamically typed?
- Weak or strong?
Given the code snippet from a fake language below:

```python
a = 10
a += 5
print(a)
```

- Explicit or implicit?
- Is it possible that the language is statically typed?
- Is it possible that the language is dynamically typed?
- Weak or strong?
Exercise: Type Systems 3

Given the code snippet from a fake language below:

```python
a = eval(user_input())
a += 5
print(a)
```

- Explicit or implicit?
- Is it possible that the language is statically typed?
- Is it possible that the language is dynamically typed?
- Weak or strong?
Exercise: Type Systems 4

Given the code snippet from a fake language below:

```
+++++++++[>++++[>+++[+<<<<-]>+>->+>+<<<<-]>+>+++
>++[-<>-]>-.>-.-------.-----+----+.>>+.>++.
<-.<.+++.-+-+--+-+---+-++.>>++.>++.
```

- Explicit or implicit?
- Is it possible that the language is statically typed?
- Is it possible that the language is dynamically typed?
- Weak or strong?
<table>
<thead>
<tr>
<th>Type System</th>
<th>Strong</th>
<th>Weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Haskell, Rust, Go</td>
<td>C, C++</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Python, Ruby, Java</td>
<td>PHP, JavaScript</td>
</tr>
</tbody>
</table>