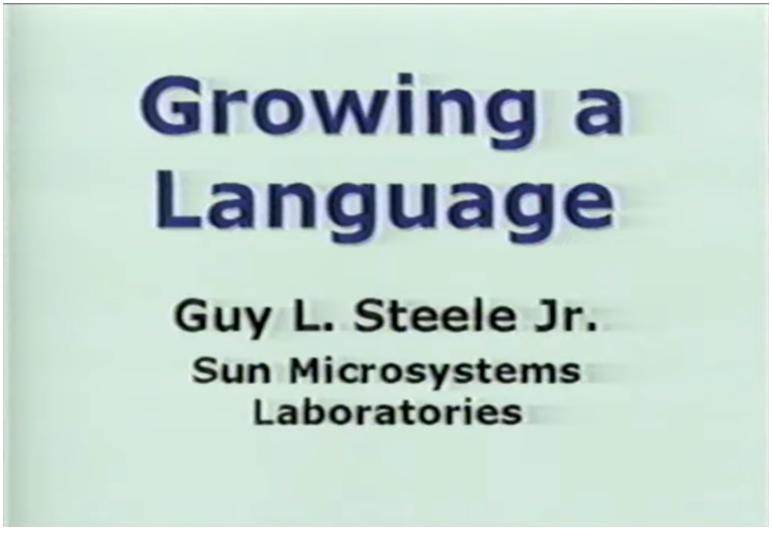
How do we grow our languages?


Growing a
Language
Guy L. Steele Jr.
Sun Microsystems
Laboratories

OOPSLA 1998

Growing a Language

I stand on this claim: I should not design a small language, and I should not design a large one. I need to design a language that can grow. I need to plan ways in which it might grow—but I need, too, to leave some choices so that other persons can make those choices at a later time.

Guy L. Steele Jr.

Growing a Language

(very reductively)

Userland *types*, *functions* and *methods* should naturally integrate into the language; as if they were part of it

A programming language should be a *pattern* for how to create a dictionary of primitives to write programs with

This applies to *all* programming languages

So what is this talk about?

So what is this talk about?

Loose bag of ideas and opinions about modern programming languages

Operator Overloading

Operator Overloading

Java BigInteger example

Operator Overloading

Java BigInteger example

This one does not spark joy

```
a.plus(b).times(c.plus(d)).divide(e)
```

Operator Overloading

Java BigInteger example

This one does not spark joy

```
a.plus(b).times(c.plus(d)).divide(e)
```

This one sparks joy

```
((a + b) * (c + d)) / e
```

Operator Overloading

Java BigInteger example

This one does not spark joy

```
a.plus(b).times(c.plus(d)).divide(e)
```

This one sparks joy

```
((a + b) * (c + d)) / e
```

To this day many languages do not support operator overloading

Operator Overloading

Arguments against:

- It can obfuscate logic

Operator Overloading

Arguments against:

- It can obfuscate logic
- ...?

James Arthur Gosling, Lead Designer of Java Programming Language™
I left out operator overloading as a fairly personal choice because I had seen too many people abuse it in C++.[1]

[1] `http://www.gotw.ca/publications/c_family_interview.htm`

James Arthur Gosling, Lead Designer of Java Programming Language™
I left out operator overloading as a fairly personal choice because I had seen too many people abuse it in C++.[1]

Zig Documentation
There is no operator overloading. When you see an operator in Zig, you know that it is doing something from this table, and nothing else.

[1] `http://www.gotw.ca/publications/c_family_interview.htm`

James Arthur Gosling, Lead Designer of Java Programming Language™
I left out operator overloading as a fairly personal choice because I had seen too many people abuse it in C++.[1]

Zig Documentation
There is no operator overloading. When you see an operator in Zig, you know that it is doing something from this table, and nothing else.

Odin Documentation FAQ
The design goals of Odin were explicitness and simplicity. Operator overloading is very easily abused and can be used to do many magical things. A procedure is clearer and more explicit.

[1] `http://www.gotw.ca/publications/c_family_interview.htm`

A procedure is clearer and more explicit.

```
divide(times(plus(a, b), plus(c, d)), e)
```

Zig tries to enforce no hidden behavior, including allocations

```
const BigInt = struct {
  data: []u8,

  fn add(
      self: BigInt,
      other: BigInt,
      allocator: Allocator,
  ) BigInt
}
```

But one can already hide the allocator in the structure

```
const BigInt = struct {
  data: []u8,
  allocator: Allocator,

  fn add(self: BigInt, other: BigInt) BigInt
};
```

But one can already hide the allocator in the structure

```
const BigInt = struct {
  data: []u8,
  allocator: Allocator,

  fn add(self: BigInt, other: BigInt) BigInt
};
```

So is then this

```
fn +(self: BigInt, other: BigInt) BigInt
```

so different?

And what about non-allocating number types?

```
const Rational = struct {
  numerator: i64,
  denominator: i64,
};
```

On the other hand

Programmers can definitely abuse tools in a wrong way

```
std::ostream& operator,(std::ostream& os, const DB& db) {
  return os << ","
            << db.select("database_name").from("metainfo");
}

// ...
std::cout, db;
// ...
```

But should we restrict expressivity because it could be abused?

On the other hand

Programmers can definitely abuse tools in a wrong way

```
std::ostream& operator,(std::ostream& os, const DB& db) {
  return os << ","
            << db.select("database_name").from("metainfo");
}

// ...
std::cout, db;
// ...
```

But should we restrict expressivity because it could be abused?

I don't think so

Error handling

Error handling

No error handling story — we are back to C land

Error handling - C++

Exceptions

Over abused in the standard library and by programmers

```
std::unordered_map<int, int> map;
int res = map.at(1); // throws
std::optional<int> mbyRes = map.try_at(1); // does not exist
```

Error handling - C++

Monadic errors

- `std::optional` - C++17
- `std::expected` (equivalent of `Either`, `Result`) - C++23
- Monadic operations (`and_then`, `transform`, `or_else`) - C++23

Still no first-class syntax for monadic errors

Error handling - C++

Sidenote - **Error code**

```
std::error_code ec;
std::filesystem::copy("file1", "file2", ec);
if (ec) { /* Handle error */ }
```

Error handling

Modern languages are often striving away from exceptions

Error handling - Zig

Single type signifying that error could happen

```
fn parseInt(buf: []u8) error{InvalidChar, OverFlow}!u64 {
  // ...
  if (ch < '0' or ch > '9') return error.InvalidChar;
  // ...
  if (overflow) return error.OverFlow;
  // ...
}

parseInt(buf) catch |err| // handle error
```

Error handling - Zig

Single type signifying that error could happen

```
fn parseInt(buf: []u8) error{InvalidChar, OverFlow}!u64 {
  // ...
  if (ch < '0' or ch > '9') return error.InvalidChar;
  // ...
  if (overflow) return error.OverFlow;
  // ...
}

parseInt(buf) catch |err| // handle error
```

Essentially an integer - what about payloads?
(*e.g.* where did the error happen?)

Error handling - Go

```
res, err := Process()
if err == nil {
  // Handle err
}
```

Universal pattern. Should it be part of the language?

Error handling - Go

```
res, err := Process()
if err == nil {
  // Handle err
}
```

Universal pattern. Should it be part of the language?

After so many years of trying, with three full-fledged proposals by the Go team and literally hundreds (!) of community proposals, most of them variations on a theme, all of which failed to attract sufficient (let alone overwhelming) support, the question we now face is: how to proceed? Should we proceed at all?
We think not.[2]

[2] `https://go.dev/blog/error-syntax`

Other ways languages grow

Other ways languages grow

- Extension methods
  - namespacing problem (monkey patching)

Other ways languages grow

- Extension methods
  - namespacing problem (monkey patching)
- ”The feature of month”
  - generators, async, coroutines, explicit IO, ...

Other ways languages grow

- Extension methods
  - namespacing problem (monkey patching)
- ”The feature of month”
  - generators, async, coroutines, explicit IO, ...
  - maybe there should be a universal way to express all of these?

Other ways languages grow

- Extension methods
  - namespacing problem (monkey patching)
- ”The feature of month”
  - generators, async, coroutines, explicit IO, …
  - maybe there should be a universal way to express all of these?
  - (monads, algebraic effects, capabilities, …?)

The ”good” - Scala

The ”good” - Scala

Scala is a massive language

- fully OOP, FP
- classes, traits, mixins, singletons, ADTs
- generics, variance, bounds, dependent types
- type classes, implicit parameters, type functions
- by-name parameters, lazy values, currying, monadic for-comprehension, pattern matching
- operator overloading, infix methods, extension methods
- experimental capture types

(non-exhaustive list)

The ”good” - Scala

Yet it does not feel unmanageable

- It evolved over time
- Features are regularly updated, deprecated and removed
- New features are added first as experimental
- The language has a vision[3]

[3] Martin Odersky has vision

The ”good” - Scala

Programmers can easily extend the language
(operator overloading, extension methods)

The ”good” - Scala

Programmers can easily extend the language
(operator overloading, extension methods)

Any userland type can do what a built-in type can*

The ”good” - Scala

Programmers can easily extend the language
(operator overloading, extension methods)

Any userland type can do what a built-in type can*

* apart from not being reference-based

The ”good” - Scala

On the other hand

The bad - C++

The bad - C++

Another massive language with tons of new features with every release

- concepts
- compiletime programming (`constexpr`, `constinit`)
- three-way comparisons
- modules
- coroutines
- receivers and senders
- metaprogramming (first part)

The bad - C++

Another massive language with tons of new features with every release

- concepts
- compiletime programming (`constexpr`, `constinit`)
- three-way comparisons
- modules
- coroutines
- receivers and senders
- metaprogramming (first part)

Yet they often feel very half-assed
Often it is a band-aid on top of band-aid

The ugly - The Truth

There is never going to be a perfect language

There is never going to be a perfect language

But we have to try out best

There is never going to be a perfect language

But we have to try out best

Or maybe programming languages are obsolete because of AI

probably not

In Summary

Go watch *Growing a Language* talk if you haven’t seen it!

Go watch *Growing a Language* talk if you haven't seen it!

Thank you!