Gleam: type-safe programming on the BEAM

Leah Neukirchen <leah@vuxu.org>

Aktiv Symposium München ’23
May 27, 2023
This is meant to be a workshop, so you’re invited to write some code. To participate, you’ll need one of these:

- A system with Linux/MacOS/WSL and having Erlang and Gleam installed on your own. Binaries are at https://github.com/gleam-lang/gleam/releases. Your distribution probably has Erlang.
- A GitHub account and a browser. Open the codespace at https://github.com/leahneukirchen/gleam-codespace.
- A system with Docker and x86_64.
  github.com/leahneukirchen/gleam-codespace/blob/main/.devcontainer/Dockerfile
What is Gleam?

A strict, functional statically-typed language with multiple backends:

• Strict means all arguments are evaluated before a call.
• Functional means values are immutable:
  • To the extent that Gleam itself does not have any global state mutation (without using FFI)
  • However, functions are not enforced to be pure
  • Perhaps there will be an effect system at a later point
• Statically typed means we don’t have null pointer exceptions or any types: well-typed programs can’t go wrong!
Gleam backends

The Gleam compiler targets the following runtimes:

• Erlang BEAM/ERTS (primary target)
  • Great support for concurrency, distribution and fault tolerance
  • Uses green threads and the actor model
• Javascript
  • Node.js
  • Deno
  • Browsers
• Perhaps some day a native compiler?
Show me some code

// comments
"Strings are UTF-8 and have C like escapes\n"
True && False // are booleans

1 + 2 * 4 == 9 // Ints and Floats are disjoint and
1.0 +. 2.5 * . 4.0 >. 9.0 // use different operators

let x = 6 // local variable bindings
[1,2,3] // lists (all elements same type)
[first, ..rest] // cons-syntax
let t = #(42,"foo",True) // tuples (various types)
t.0 == 42
Blocks

Instead of parentheses, blocks surrounded by curly braces are used for grouping:

```
{1 + 2} * 4 == 12
```

```
{
  let a = 1 + 2
  let b = 4
  a * b
} == 12
```
Gleam features full type inference, i.e. all type annotations are optional (but will be checked).

let x: Int = 6

Gleam uses a *Hindley-Milner type system*: bleeding-edge from the 1960s!
Pattern matching

```rust
case myresult {
    Ok(x) -> x
    Error(e) -> panic
}
let assert Ok(x) = myresult  // single case

case some_bool {
    True -> "It's true!"
    False -> "It's not true."
}
```

Gleam supports list patterns, string prefix matches, guards, and matching multiple values at once.
Functions

```rust
fn add(x: Int, y: Int) -> Int {
    x + y
}
fn add(x, y) {
    x + y
}
fn map(list: List(a), fun: fn(a) -> b, acc: List(b)) -> List(b) {
    case list {
        [] -> reverse(acc)
        [x, ..xs] -> map(xs, fun, [fun(x), ..acc])
    }
}
```

You can use tail-recursion for self calls.

Leah Neukirchen: Gleam
More functions

Functions can have labelled arguments:

```rust
pub fn replace(
    in string: String,
    each pattern: String,
    with replacement: String,
) {
    // Code with string, pattern, and replacement
}

replace(each: ",", with: " ", in: "A,B,C")

let add_one = add(1, _) // currying
add_one(2)
```
The pipe operator

Instead of deeply nesting calls a(b(c(d))), Gleam supports the pipe operator |>:

d |> c |> b |> a

You can pass additional arguments to the functions:

items
|> list.reverse
|> list.map(fn(arg) { " " <> arg })
|> string_builder.from_strings

Gleam will figure out *from the types* which argument is to be used for the pipeline!
Modules

Every file is a module, determined by its file name.

You need to mark functions to be exported as `pub fn`.

```rust
import unix/cat
import animal/cat as kitty
import animal/cat.{Cat, stroke}
```
Custom types

You can define your own algebraic data types:

type User {
    LoggedIn(name: String)
    Guest
}

Leah Neukirchen: Gleam
Orthography

Modules, variables, constants and functions are lowercase_and_snake_case.

Types and their constructors are CamelCase.

This is enforced by the compiler.

Project names (= top level directories) must start with a lowercase letter and may only contain lowercase letters, numbers and underscores. Project names cannot start with gleam_.

Leah Neukirchen: Gleam
Standard library types

The prelude contains

BitString
Bool
Float
Int
List(element)
Nil
Result(value, error)
String
UtfCodepoint
The Result type

type Result(value, reason) {
  Ok(value)
  Error(reason)
}

Haskellers know this as Either.

To Rustaceans this should be familiar.
Use notation

use notation passes the rest of the code of the block as a function:

Instead of the “pyramid of doom” as in

```javascript
logger.record_timing(fn() {
  database.connect(fn(db) {
    file.open("file.txt", fn(f) {
      // Do something with `f` here...
      
    })
  })
})
```

Leah Neukirchen: Gleam
Use notation, cont’d

We can write:

```r
use <- logger.record_timing
use db <- database.connect
use f <- file.open("file.txt")
// Do something with `f` here...
```
Topics not covered

- Erlang/JS FFI
- Bit strings

That is pretty much everything you need to learn about the language!

Gleam favors first-class-functions and data types over many features other languages have.

https://mckayla.blog/posts/all-you-need-is-data-and-functions.html
Tooling

• `gleam new` creates a new Gleam project
• `gleam run` compiles and starts the project
• `gleam test` compiles and runs the tests
• `gleam add` adds a dependency to a project
• `gleam format` reformatst the code (quite opinionated...)
• `gleam fix` to apply language syntax changes
• `gleam lsp` starts the included LSP server
Caveats

• Some quirks require some Erlang background to understand.
  • e.g. what you can call in an “if”-guard
  • string functions and graphemes
• Some things are inconsistent between backends:
  • Erlang has big integers, JavaScript only 32-bit
  • Erlang has full tail call optimization, JavaScript only for self-calls
• Type **errors** result in spurious warnings, which are printed first, but are often wrong.
• No REPL, but the compiler is quick.
Questions?

Thank you.

Now let’s hack on something!