Gleam Syntax

Comments

Comments must start with forward slashes.

```
// This is a comment in Gleam

/// This is comment to document a statement
const answer: Int = 42

//// This is comment to document a module
```

Operators

In Gleam the following operators are available:

```
==  // Equal
!=  // Not equal
>   // Int greater than
>.  // Float greater than
>=  // Int greater or equal
>=. // Float greater or equal
<   // Int less than
<.  // Float less than
>=  // Int less or equal
>=. // Less or equal
&&  // Boolean and
||  // Boolean or
+   // Int add
+.  // Float add
-   // Int subtract
-.  // Float subtract
*   // Int multiply
*.  // Float multiply
/   // Int divide
/.  // Float divide
%   // Modulo
|>  // Pipe
```

Variables & Data Types

The `let` keyword is used before variable names.

```
let size = 5_000

// Re-assignment of the 'size' variable
let size = size + 1_000
let size = 1_000_000
```

Variable type annotations

Type annotations can be given when binding variables.

```
let some_float: Float = 1.0
let some_int: Int = 1

// A list contains elements of the same data type
let some_list: List(Int) = [1, 2, 3]

// Strings are UTF-8 encoded and use double quotes
let some_string: String = "Hellø, world!"

// Tuples allow elements with different data types
let some_tuple: #(Int, Float, String) = #(1, 1.0, "1")
```

Maps

Maps can have keys and values of any type, but all keys must be of the same type.

```
import gleam/map

map.from_list([#("key1", "val1"), #("key2", "val2")])
```

Furthermore, all values must be of the same type otherwise:

```
import gleam/map

// Type error!
map.from_list([#("key1", "val1"), #("key2", 2)])
```

There is no map literal syntax in Gleam, and you cannot pattern match on a map. Maps are generally not used much in Gleam.

Match operator

The `let` keyword and the `=` operator can be used for pattern matching. For assertions, the `assert` keyword is preferred.

```
// Bind element 1 in tuple to name x
let #(x, _) = #(1, 2)

// Pattern match with 1 in list and bind 2 to name y
let [1, y, _] = [1, 2, 3]

// Runtime error
assert [] = [1]

// Compile error, type mismatch
assert [z] = "Hello"
```

For list destructuring and pattern matching the `..` operator can be used.

```
let list = [2, 3, 4]

// Prepend element 1 to the list
let list = [1, .. list]

// Pattern match with 1 in the list and bind the
// second element to name 'second_element'
let [1, second_element, .. ] = list
```

Functions

Functions are declared using the `fn` keyword.

```
// This is a named function
fn sum(x, y) {
  x + y
}

// This is an anonymous function
let mul = fn(x, y) { x * y }
mul(1, 2)
```

Exporting functions

Functions are private by default but can be made public using the `pub` keyword.

```
// This is a private function
fn mul(x, y) {
  x * y
}

// This is a public function
pub fn sum(x, y) {
  x + y
}
```

Function type annotations

Functions can have their argument and return types annotated.

```
pub fn add(x: Int, y: Int) -> Int {
  x + y
}

// Compile error, return type mismatch
pub fn mul(x: Int, y: Int) -> Bool {
  x * y
}
```

Labelled function arguments

Arguments can be given a label as well as an internal name.

```
pub fn replace(in string, each pattern, with sub) {
  some_func(string, pattern, sub)
}

// Note: Labelled arguments can be given in any order
replace(each: ",", with: " ", in: "A,B,C")
```

Referencing functions

A function can be referenced without any special syntax.

```
// Function to be referenced
pub fn identity(x) {
  x
}

// Assignment of the function to a variable
let func = identity
func(100)
```

Constants

Constants can be created using the `const` keyword.

```
const the_answer = 42

// We can return the value through a function call
pub fn main() {
  the_answer
}

// Constants can also be exported and referenced in
// other modules if declared with the 'pub' keyword
pub const other_answer = 99
```

Expression blocks

Braces { } are used to group expressions.

```
pub fn main() {
  let x = {
    some_func(1)
    2
  }
  // Braces change the order of arithmetic operations
  let y = x * {x + 10}
  y
}
```

Flow control

The `case` expression provides a way to match a value type to an expression. Some examples are given in the following.

```
case some_number {
  0 -> "Zero"
  1 -> "One"
  2 -> "Two"
  // This matches anything
  n -> "Some other number"
}
```

A `case` expression can be coupled with destructuring to provide native pattern matching.

```
case xs {
  [] -> "This list is empty"
  [a] -> "This list has 1 element"
  [a, b] -> "This list has 2 elements"
  _other -> "This list has more than 2 elements"
}
```

A `case` expression also supports guards.

```
case xs {
  // Use the 'if' keyword to add a guard expression
  // to the case clause
  [a, b, c] if a >. b && a <=. c -> "ok"
  _other -> "ko"
}
```

...along with disjoint union matching.

```
case number {
  2 | 4 | 6 | 8 -> "This is an even number"
  1 | 3 | 5 | 7 -> "This is an odd number"
  _ -> "I'm not sure"
}
```

Error management

Handling errors means to match the return value of an operation against an `Error` or `Ok` container.

```
case parse_int("123") {
  Error(e) -> io.println("That wasn't an Int")
  Ok(i) -> io.println("We parsed the Int")
}
```

The `try` keyword simplifies this. It will either bind a value to the providing name if `Ok(Value)` is matched, or interrupt the flow and return `Error(Value)`. An example is given in the following.

```
let a_number = "1"
let an_error = Error("ouch")
let another_number = "3"

// Successful
try int_a_number = parse_int(a_number)

// Error will be returned
try attempt_int = parse_int(an_error)

// Consequently, this never gets executed
try int_another_number = parse_int(another_number)

// ... and this never gets executed
Ok(int_a_number + attempt_int + int_another_number)
```

Type aliases

Type aliases allow for referencing of arbitrary complex types. Even though their type systems do not serve the same function. The `type` keyword can be used to create aliases.

```
pub type Headers =
  List(#(String, String))
```

Custom types

Records

A Custom type allows you to define a collection data type with a fixed number of named fields, and the values in those fields can be of differing types.

```
type Person {
  Person(name: String, age: Int)
}

let person = Person(name: "Jake", age: 35)
let name = person.name
```

Unions

Functions must always take and receive one type. To have a union of two different types they must be wrapped in a new custom type.

```
type IntOrFloat {
  AnInt(Int)
  AFloat(Float)
}

fn int_or_float(X) {
  case X {
    True -> AnInt(1)
    False -> AFloat(1.0)
  }
}
```

Opaque custom types

Custom types can be defined as being opaque, which causes the constructors for the custom type not to be exported from the module. Without any constructors to import other modules can only interact with opaque types using the intended API.

```
pub opaque type Identifier {
  Identifier(Int)
}

// Return 'Identifier' with an assigned value of 100
pub fn get_id() {
  Identifier(100)
}
```

Modules

A Gleam file is a module that is named based on its filename and directory path.

```
// In file foo.gleam
pub fn identity(x) {
  x
}

// In file main.gleam
import foo

pub fn main() {
  foo.identity(1)
}
```

Imports

Imports are relative to the root src folder. Modules in the same directory will need to reference the entire path from src for the target module, even if the target module is in the same folder.

```
// Inside src/nasa/moon_base.gleam
// Imports src/nasa/rocket_ship.gleam
import nasa/rocket_ship

pub fn explore_space() {
  rocket_ship.launch()
}
```

Named Imports

Named imports can be accomplished using the **as** keyword.

```
import unix/cat
import animal/cat as kitty
```

Unqualified imports

Unqualified imports are done in the following way:

```
import animal/cat.{Cat, stroke}

pub fn main() {
  let kitty = Cat(name: "Nubi")
  stroke(kitty)
}
```

Read the documentation

Link to gleam.run/documentation/

Gleam v0.16.1 Syntax Cheatsheet v0.1.0. Typeset using XeTeX by NicklasXYZ, August 15, 2021.
Source code available on GitHub. Released under an Apache License 2.0.