Interpreters

Announcements

External Functions

External Functions

Gleam compiles to JavaScript, which allows mutation

JavaScript functions can be called from Gleam, and those functions can return Gleam values

Therefore, it's possible to create a mutable Gleam object!

```gleam
pub type Cache
@external(javascript, "./fib_ffi.mjs", "new_cache")
pub fn new_cache() -> Cache
@external(javascript, "./fib_ffi.mjs", "lookup")
fn lookup(cache: Cache, n: Int) -> Result(Int, Nil)
@external(javascript, "./fib_ffi.mjs", "store")
fn store(cache: Cache, n: Int, value: Int) -> Int
pub fn main() {
  let cache = new_cache()
  store(cache, 5, 25)
  let result = case lookup(cache, 5) {
    Ok(v) -> v
    Error(Nil) -> 0
  }
}
```

Gleam type name for a JavaScript object

Mutation!

result is assigned 25

```javascript
import { Ok, Error } from "./gleam.mjs";
export function new_cache() {
  return new Map();
}
export function lookup(cache, n) {
  return cache.has(n)
    ? new Ok(cache.get(n))
    : new Error(undefined);
}
export function store(cache, n, value) {
  cache.set(n, value);
  return value;
}
```

Javascript's dict

Javascript's if/else

Exercise: Count the Number of Function Calls in fib(42)

```gleam
pub type Counter
@external(javascript, "./fib_ffi.mjs", "new_counter")
pub fn new_counter() -> Counter
@external(javascript, "./fib_ffi.mjs", "increment")
fn increment(counter: Counter) -> Nil
@external(javascript, "./fib_ffi.mjs", "count")
pub fn count(counter: Counter) -> Int
pub fn fib(n: Int, counter: Counter) -> ___________ {
  ____________________________________________________
  case n {
    0 | 1 -> ________________________________________
    _ -> ____________________________________________
  }
}
pub fn main() {
  let counter = new_counter()
  ____________________________________________________
  echo _______________ // echo number of calls in fib(42)
}
```

```gleam
/// The nth Fibonacci number.
pub fn fib(n: Int) -> Int {
  case n {
    0 | 1 -> n
    _ -> fib(n - 2) + fib(n - 1)
  }
}
```

Creates an object with an attribute named "calls" that has value 0

```javascript
export function new_counter() {
  return { calls: 0 };
}

export function increment(counter) {
  counter.calls += 1;
}

export function count(counter) {
  return counter.calls;
}
```

Exercise: Count the Number of Function Calls in fib(42)

```gleam
pub type Counter
@external(javascript, "./fib_ffi.mjs", "new_counter")
pub fn new_counter() -> Counter
@external(javascript, "./fib_ffi.mjs", "increment")
fn increment(counter: Counter) -> Nil
@external(javascript, "./fib_ffi.mjs", "count")
pub fn count(counter: Counter) -> Int
pub fn fib(n: Int, counter: Counter) -> Int {
  increment(counter)
  case n {
    0 | 1 -> n
    _ -> fib(n - 2, counter) + fib(n - 1, counter)
  }
}
pub fn main() {
  let counter = new_counter()
  fib(42, counter)
  echo count(counter) // echo number of calls in fib(42)
}
```

```gleam
/// The nth Fibonacci number.
pub fn fib(n: Int) -> Int {
  case n {
    0 | 1 -> n
    _ -> fib(n - 2) + fib(n - 1)
  }
}
```

Creates an object with an attribute named "calls" that has value 0

```javascript
export function new_counter() {
  return { calls: 0 };
}

export function increment(counter) {
  counter.calls += 1;
}

export function count(counter) {
  return counter.calls;
}
```

Animator Project

(Demo)

Two Interpreters

Calculator (Homework 8, assigned Friday):

- Represent integer arithmetic with variables: `(x + 3) * 5`
- Define & apply anonymous functions: `fn(x, y) { 2 * y - x }`

Evaluator (Animator Project):

- An environment is a sequence of frames
- User-defined functions can be recursive
- More built-in functions and operators (beyond +, -, *, /)
- More types of expressions: case, let, dot expressions (field access)
- More types of values: bools, lists, functions, records

Final Exam Questions to Expect

How is the Gleam expression `1+1` represented in the Animator project's evaluator?

How is the Gleam value `[1, 2]` represented in the Animator project's evaluator?

What expressions are evaluated while evaluating `[y + 2]` in the Animator project's evaluator?

Parsing Gleam

Parsing and Abstract Syntax Trees

The abstract syntax tree for some expression in a program is represented by an **Expr** value

A simple but useful subset of Gleam:

Example expressions:
`x`
`True`
`61`
`[1, 2] or [1, 2, ..s]`
`f(5)`
`fn (x) {x > 5}`
`case x {0 -> False _ -> True}`
`let x = 5`
`point.x`

```gleam
pub type Expr {
  Var(String)
  LiteralBool(Bool)
  LiteralInt(Int)
  LiteralList(elems: List(Expr), rest: Option(Expr))
  Call(fn_expr: Expr, operands: List(Expr))
  Lambda(params: List(String), body: Expr)
  Case(subject: Expr, arms: List(Arm))
  Let(name: String, value: Expr, body: Expr)
  FieldAccess(record: Expr, field: String)
}
```

An "arm" of a case expression

The expression on the line after the **let** line

(Demo)

Eval & Apply

Representing Values

Evaluating an expression in an environment produces a value (or an evaluation error)

Example values:

```
True
61
[1, 2]
Position(5, 10)
fn (x) {x > 5}
```

```gleam
pub type Value {
  VBool(Bool)
  VInt(Int)
  VList(List(Value))
  VData(tag: String, fields: List(Value))
  VFunc(Callable)
}
```

A **Builtin** function calls some existing Gleam function

```gleam
pub type Callable {
  Builtin(name: String, num_args: Int, run: fn(List(Value)) -> Result(Value, EvalError))
  UserFn(name: String, params: List(String), body: Expr, env: Env)
}
```

A **UserFn** has a body **Expr** that is evaluated

The environment in which it was defined

Gleam Evaluation

```
fn eval(expr: Expr, env: Env) -> Result(Value, EvalError) {
  case expr {
    Var(name) -> lookup(env, name)
    LiteralBool(b) -> Ok(VBool(b))
    LiteralInt(n) -> Ok(VInt(n))
    LiteralList(elems, rest) ->   // Evaluate & combine the elems and rest
    Call(fn_expr, operands) ->    // Evaluate the fn_expr & operands, then call apply
    Lambda(params, body) ->       // Construct a UserFn("lambda", ...)
    Case(s, arms) ->              // Evaluate s, then match its value to one of the arms
    Let(x, v, body) ->            // Evaluate v, bind its value to x, and then evaluate body
    FieldAccess(record, field) -> // Evaluate record, make sure it's a VData, and get a field
  }
}
pub fn apply(f: Value, args: List(Value)) -> Result(Value, EvalError) {
  case f {
    VFunc(Builtin(name, num_args, run)) ->        // Call run (a Gleam function) on args
    VFunc(UserFn(name, params, body, parent)) ->  // Make a Local frame e, then eval(body, e)
  }
}
```

Evaluation Demo

What happen when we evaluate f([1, 2]) for this f?

```
fn f(s) {
  case s {
    [] -> [3]
    [first, ..rest] -> [first, ..f(rest)]
  }
}
```

To see what happens during evaluation, we can

- Print each expression being evaluated
- Print each function being applied and its arguments
- Print again each time a function returns a value

(Demo)