Immutable Data

Types Review

Non-Empty Lists

Try to spot the bug in this implementation of **increasing** for non-empty linked lists

```
pub type NotEmpty(a) {
  Last(first: a)
  Link(first: a, rest: NotEmpty(a))
}

pub fn starts_with(s: NotEmpty(a), v: a) -> Bool {
  s.first == v
}

pub fn increasing(s: NotEmpty(Int)) -> Bool {
  case s {
    Last(_) -> True
    Link(first, Link(second, _)) if second <= first -> False
    Link(_, rest) -> increasing(rest)
  }
}
```

https://is.gd/61a_debug_increasing

Multiple Subjects

A Case Expression Can Have Multiple Subjects

```gleam
import gleam/int

pub fn main() {
  let x = int.random(2)
  let y = int.random(2)
  echo x
  echo y

  let result = case x, y {
    0, 0 -> "Both are zero"
    0, _ -> "First is zero"
    _, 0 -> "Second is zero"
    _, _ -> "Neither are zero"
  }
  echo result
}
```

Patterns match combinations of a value for x and a value for y

https://tour.gleam.run/flow-control/multiple-subjects/

Multiple Subjects is Equivalent to Tuple Pattern Matching

```
pu
warning: Redundant tuple
   ┌─ /src/main.gleam:9:21
   │
9  │   let result = case #(x, y) {
   │                     ^^^^^^^ You can remove this tuple wrapper

Instead of building a tuple and matching on it, you can match on its
contents directly.
A case expression can take multiple subjects separated by commas like
this:

}
    case one_subject, another_subject {
      _, _ -> todo
    }

See: https://tour.gleam.run/flow-control/multiple-subjects/
```

Example: Matching Lists

Lists s and t match if they are equal or one is a prefix of the other

```
assert matches([1, 2, 3], [1, 2, 3])
assert matches([1, 2, 1, 2], [1, 2])
assert matches([1, 2], [1, 2, 3, 4])
assert matches([], [1, 2])
assert !matches([1, 2, 3], [1, 3])

fn matches(s: List(a), t: List(a)) -> Bool {
  case s, t {
    ____________________________________________________________________
    _, _ -> True
  }
}
```

Playground link: is.gd/61a_matches

Example: Matching Lists

Lists s and t match if they are equal or one is a prefix of the other

```
assert matches([1, 2, 3], [1, 2, 3])
assert matches([1, 2, 1, 2], [1, 2])
assert matches([1, 2], [1, 2, 3, 4])
assert matches([], [1, 2])
assert !matches([1, 2, 3], [1, 3])

fn matches(s: List(a), t: List(a)) -> Bool {
  case s, t {
    [x, ..s_rest], [y, ..t_rest] -> x == y && matches(s_rest, t_rest)
    _, _ -> True
  }
}
```

This pattern matches any value of s and any value of t

Playground link: is.gd/61a_matches

Example: Repeating Lists Using Matching

Return whether a list s is n values repeated; for n==3 [x, y, z, x, y, z, x, y, z, x, ...]
[x, y, z, x, y, z, x, ...]

```
pub fn repeating(s: List(a), n: Int) -> Bool {
  matches(s, _______(s, n))
}

/// ____________________________________

fn _____(s: List(a), n: Int) -> List(a) {

}
```

Lists s and t match if they are equal or one is a prefix of the other

Playground link: is.gd/61a_repeating2

Example: Repeating Lists Using Matching

Return whether a list s is n values repeated; for n==3 [x, y, z, x, y, z, x, y, z, x, ...]
[x, y, z, x, y, z, x, ...]

```
pub fn repeating(s: List(a), n: Int) -> Bool {
  matches(s, drop(s, n))
}
```

Lists s and t match if they are equal or one is a prefix of the other

```
/// Remove up to the first n items of s
fn drop(s: List(a), n: Int) -> List(a) {
  case n {
    0 -> s
    _ ->
      case s {
        [] -> []
        [_, ..rest] -> drop(rest, n - 1)
      }
  }
}
```

Playground link: is.gd/61a_repeating2

Results

Results

The built-in Result type is used for value-based error handling

In Python, most code is written assuming no errors occur, and error handling is optional

```
>>> s = [3, 4]
>>> s[5]
Traceback (most recent call last):
  File "<python-input-2>", line 1, in <module>
    s[5]
    ~^^^
IndexError: list index out of range
```

Gleam programs do not run until the compiler confirms that all errors are handled

```
fn first_int(s: List(Int)) -> Int {
  case s {
    [x, ..] -> x
    [] -> 0
  }
}
```

```
fn nth_int(s: List(Int), n: Int) -> Int {
  case list.first(list.drop(s, n)) {
    Ok(k) -> k
    Error(Nil) -> 0
  }
}
```

This expression evaluates to a Result(Int, Nil)

https://tour.gleam.run/data-types/results/

Creating and Processing Results

To create a result, write `Ok(...)` or `Error(...)`

To process a result, write:

```
case some_expression_that_evaluates_to_a_result {
  Ok(...) -> ...
  Error(...) -> ...
}
```

There are convenience functions for working with results in the Gleam standard library

`list.try_map` takes a function that returns a **`Result`** and applies it to each item in a list, returning a list of **`Ok`** values or the first **`Error`**.

`int.parse` has the signature: **`pub fn parse`**`(string: String) -> Result(Int, Nil)`

```
list.try_map(["1", "2", "3"], int.parse)    // Evaluates to Ok([1, 2, 3])
list.try_map(["1", "two", "3"], int.parse)  // Evaluates to Error(Nil) because of "two"
```

Example: an Item of a List

Implement **nth**, which takes a list s and an integer n. It returns the nth item of s.

```
fn nth(s: List(a), n: Int) -> Result(a, Nil) {
  case s, n {
    [], _ -> Error(Nil)
    _, _ if n < 0 -> Error(Nil)
    ____________________________________
    ____________________________________
  }
}
```

Example: an Item of a List

Implement **nth**, which takes a list s and an integer n. It returns the nth item of s.

```
fn nth(s: List(a), n: Int) -> Result(a, Nil) {
  case s, n {
    [], _ -> Error(Nil)
    _, _ if n < 0 -> Error(Nil)
    [x, ..], 0 -> Ok(x)
    [_, ..rest], _ -> nth(rest, n - 1)
  }
}
```

Why didn't we need to write Ok(...) or Error(...)?

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) -> 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;
}
```

Immutable Data

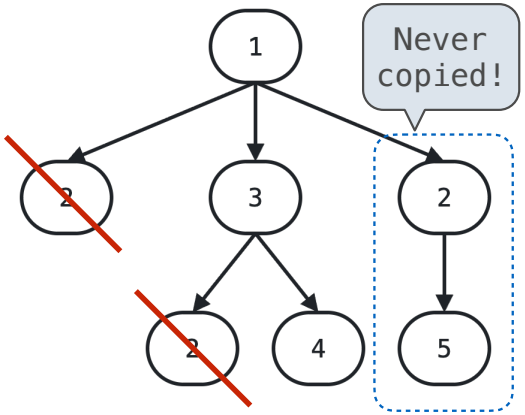
Exercise: Trees with Shared Subtrees

Implement remove_leaf, which removes the leftmost leaf of a tree unless the tree is a leaf

```
pub type Tree(a) {
  Tree(label: a,
       branches: List(Tree(a)))
}

pub fn leaf(label: a) -> Tree(a) {
  Tree(label, [])
}
```

```
remove_leaf(
  remove_leaf(
    Tree(1, [
      leaf(2),
      Tree(3, [leaf(2), leaf(4)]),
      Tree(2, [leaf(5)])]) ) )
```




```
/// A new tree without its leftmost leaf; a tree that is a leaf is unchanged
pub fn remove_leaf(tree: Tree(a)) -> Tree(a) {
  case tree {

    Tree(label, _____________________________) -> Tree(label, rest)

    Tree(label, [first, ..rest]) -> ____________________________________________________

    Tree(_, []) -> tree
  }
}
```

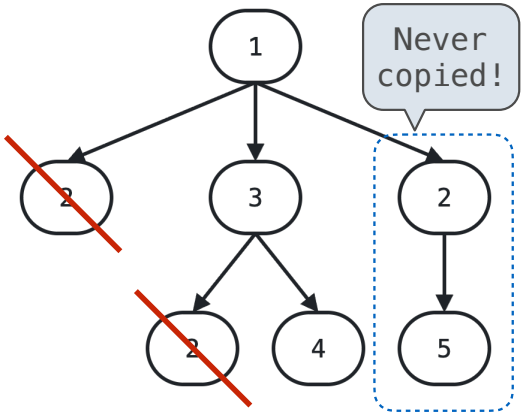
Exercise: Trees with Shared Subtrees

Implement remove_leaf, which removes the leftmost leaf of a tree unless the tree is a leaf

```
pub type Tree(a) {
  Tree(label: a,
       branches: List(Tree(a)))
}

pub fn leaf(label: a) -> Tree(a) {
  Tree(label, [])
}
```

```
remove_leaf(
  remove_leaf(
    Tree(1, [
      leaf(2),
      Tree(3, [leaf(2), leaf(4)]),
      Tree(2, [leaf(5)])]) ) )
```




```
/// A new tree without its leftmost leaf; a tree that is a leaf is unchanged
pub fn remove_leaf(tree: Tree(a)) -> Tree(a) {
  case tree {
    Tree(label, [Tree(_, []), ..rest]) -> Tree(label, rest)
    Tree(label, [first, ..rest]) -> Tree(label, [remove_leaf(first), ..rest])
    Tree(_, []) -> tree
  }
}
```