Algebraic Data Types

Announcements

Lists

Gleam Lists are Linked Lists

A list literal looks like a Python list: `[3, 7, 5, 9]`

However, a list literal can also contain an expression for the rest of the list

```
let s = [5, 7, 9]
let odds = [1, 3, ..s]  // Evaluates to [1, 3, 5, 7, 9], sharing memory with s
```

This `..` can only appear once, only at the end, and only before a list

All items in a Gleam list must have the same type; the following cause type mismatch errors

```
[1, "two"]
["one", ..s]
[1, 3, s]
```

A list type describes the type of the items that the list contains

```
import gleam/list
fn double_each(s: List(Int)) -> List(Int) {
  list.map(s, fn(n) { 2 * n })
}
```

The list module contains map, filter, all, any, append, zip, etc.

A list of integers

https://gleam-stdlib.hexdocs.pm/gleam/list.html#map

List Patterns

There is no .first or .rest of a Gleam list

Instead, a case expression for a list can assign names to the first item(s) and the rest

```
fn describe(s: List(Int)) -> String {
  case s {
    [] -> "Empty list"
    [1] -> "List of just 1"
    [4, ..] -> "List starting with 4"
    [a, b] if a < b -> "List of " <> int.to_string(a) <> " then " <> int.to_string(b)
    [_, _] -> "List of two elements"
    [a, b, ..rest] if a < b -> "List ending in " <> string.inspect(rest)
    _ -> "Some other list"
  }
}
```

a is now a name for the first item in s

rest is now a name for the rest of the list after the first two items (a and b)

https://playground.gleam.run/#N4IgbgpgTgzglgewHYgFwEYA0IDGyAuES+aIcAtgA4JT4AEA5gDYQCGSA9HMQDpIXVajFuw4x8Ubgz59KAVwBGdAGZI65VtwAUASjrA+dOhBwALBHQAmEGDkkKIWgNoBdHYeNmL12/cdP0Nw8TcysbOzgHZwAmILVPUJBIqKcAFkw6AFYMgDY4oxDvcL9nAGYM7LocjIB2fISi30j/
cqz6wrCmlMrWvPckAF8ZJFVO5McYVDoAGThxLQBJYh09AFoAPjoAZQkpfQ8cVhgIOhh9+KNXOg2GHhAAUSp8AE86JjnBO49LwOvNu9m4joCGUdAAVnIgegvhcGGkMgA6BEuP63ECA+jiVi0PYAdzg+FMdFSMKMl1YGQUKLgoNYdAAPHQlBtvmSAR9gaC7gzNtx8Aj8AgAPriSRIBhaVh6en/EB0QlENE8uh8gXC0VSLQKfpkuFCjJClE3dlAkHy3EWCAschEfAwUlkpwUpmIhFQGz4am0hlM1Em+hESx7bhKmWnXbihHcGCUEz4LTu8Q6slCv0gLYIG3AhVQN4fB1DQZ8EADIA

List Module

https://gleam-stdlib.hexdocs.pm/gleam/list.html

https://playground.gleam.run/#N4IgbgpgTgzglgewHYgFwEYA0IDGyAuES+aIcAtgA4JT4AEA5gDYQCG5A9HMQDpIXVajFuw5M4MfHz6UArgCM6AMyR1yrbgAoAlHWB86dFvRh0AvHQDaAZkx0ALHYCsdgGx2A7AF1pquhw4jCXwAOnVK0lZKSnEIU1ZlWSQcfERVfAQ6NhwACyyWciIpPwhczPFJMKjNGDsVTQAPXWA6BroAK1a6AF9tX0MAoMqlOCZCKDoAawgISlNciHyIQuJTJRo6AHccuFzIxOTU5DooCHxZKCRTABUoWQgDLLKh0JGx6Bq7blCJAH0ISBIPolZ4VV6jcafZRIRrNLoAPgcPWB3RA3SAA===

List Processing Examples

Example: Alternating Lists

Return whether a list s contains a pair of alternating items [x, y, x, y, x, ...]

Gleam feature: two values of the same type can always be compared using == and !=
(There is no equivalent of Python's **is** in Gleam because all values are immutable)

Gleam syntax: The equivalent of Python's **and** is **&&**, **or** is **||**, **not** is **!**

```
fn alternating(s: List(a)) -> Bool {
  case s {
    _____________________________ -> True
    _______________ if _______ -> False
    _____________ -> alternating(rest)
  }
}
```

```
assert alternating([])
assert alternating([1, 2, 1, 2, 1])
assert alternating(["a", "b", "a", "b"])
assert !alternating([1, 2, 1, 3])
assert !alternating([1, 2, 2, 1])
```

Playground link: is.gd/61a_alternating

Example: Alternating Lists

Return whether a list s contains a pair of alternating items [x, y, x, y, x, ...]

Gleam feature: two values of the same type can always be compared using == and !=
(There is no equivalent of Python's **is** in Gleam because all values are immutable)

Gleam syntax: The equivalent of Python's **and** is **&&**, **or** is **||**, **not** is **!**

```
fn alternating(s: List(a)) -> Bool {
  case s {
    [] | [_] | [_, _]          -> True
    [x, _, z, ..] if x != z    -> False
    [_, ..rest]                -> alternating(rest)
  }
}
```

```
assert alternating([])
assert alternating([1, 2, 1, 2, 1])
assert alternating(["a", "b", "a", "b"])
assert !alternating([1, 2, 1, 3])
assert !alternating([1, 2, 2, 1])
```

Playground link: is.gd/61a_alternating

Example: Repeating Lists

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

Gleam syntax: The equivalent of Python's **and** is **&&**, **or** is **||**, **not** is **!**

```
pub fn repeating(s: List(a), n: Int) -> Bool {
  case s {
    [] -> True
    [first, ..rest] -> nth_equals(first, s, n) && repeating(rest, n)
  }
}

/// Return True if s is shorter than n or s[n] == expected
pub fn nth_equals(expected: a, s: List(a), n: Int) -> Bool {
  case s {
    [] -> True
    
  }
}
```

Playground link: is.gd/61a_repeating

Example: Repeating Lists

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

Gleam syntax: The equivalent of Python's **and** is **&&**, **or** is **||**, **not** is **!**

```
pub fn repeating(s: List(a), n: Int) -> Bool {
  case s {
    [] -> True
    [first, ..rest] -> nth_equals(first, s, n) && repeating(rest, n)
  }
}

/// Return True if s is shorter than n or s[n] == expected
pub fn nth_equals(expected: a, s: List(a), n: Int) -> Bool {
  case s {
    [] -> True
    [first, ..rest] ->
          case n {
            0 -> first == expected
            _ -> nth_equals(expected, rest, n - 1)
          }
  }
}
```

Playground link: is.gd/61a_repeating

Example: Matching Lists

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

Gleam syntax: A case expression can have multiple subjects

```gleam
  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, 2])

fn matches(s: List(a), t: List(a)) -> Bool {
  case s, t {

    _, _ -> True
  }
}
```

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

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

Gleam syntax: A case expression can have multiple subjects

```
  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))
}

/// 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)
      }
  }

}
```

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

Playground link: is.gd/61a_repeating2

Algebraic Data Types

Patterns and Types

A **case** pattern gives local names to the fields (also "attributes") of each variant

```
pub type Contact {
  EmailOnly(email: String)
  PhoneOnly(phone: Int)
  Both(email: String, phone: Int)
}

pub type User {
  User(name: String, contact: Contact)
}

pub fn contact(user: User) -> String {
  case user.contact {
    EmailOnly(e) | Both(e, _) -> "mailto:" <> e
    PhoneOnly(n) -> "tel:" <> int.to_string(n)
  }
}
```

```
contact(User("Oski", EmailOnly("oski@berkeley.edu")))

"mailto:oski@berkeley.edu"
```

Patterns and Types

A **case** pattern gives local names to the fields (also "attributes") of each variant

Attributes can also be accessed through dot expressions *after* matching a variant

```
pub type Contact {
  EmailOnly(email: String)
  PhoneOnly(phone: Int)
  Both(email: String, phone: Int)
}

pub type User {
  User(name: String, contact: Contact)
}

pub fn contact(user: User) -> String {
  case user.contact {
    EmailOnly(..) as c -> "mailto:" <> c.email
    Both(..) as c -> "mailto:" <> c.email
    PhoneOnly(..) as c -> "tel:" <> int.to_string(c.phone)
  }
}
```

user.contact.email is not allowed because user.contact may not have an email attribtue

Type Examples

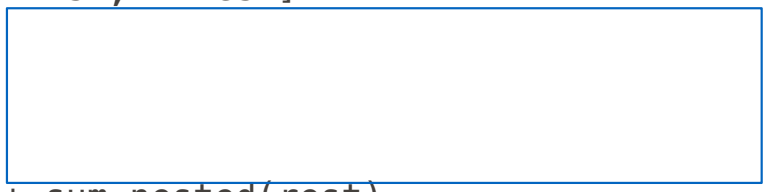
Nested Lists

A nested list of integers is a list containing integers and nested lists of integers

```gleam
pub type NestedList(a) =
  List(Nested(a))

pub type Nested(a) {
  Sublist(NestedList(a))
  Item(a)
}
```

```gleam
// Sum a nested list of integers
pub fn sum_nested(s: NestedList(Int)) -> Int {
  case s {
    [] -> 0
    [first, ..rest] ->
      
      + sum_nested(rest)
  }
}
```



```gleam
// Sum a nested list of integers with list.map
pub fn sum_nested(s: NestedList(Int)) -> Int {
  int.sum(
    
  )
}
```



https://is.gd/61a_sumnested

Nested Lists

A nested list of integers is a list containing integers and nested lists of integers

```
pub type NestedList(a) =
  List(Nested(a))

pub type Nested(a) {
  Sublist(NestedList(a))
  Item(a)
}
```

```
// Sum a nested list of integers
pub fn sum_nested(s: NestedList(Int)) -> Int {
  case s {
    [] -> 0
    [first, ..rest] ->
      case first {
        Sublist(t) -> sum_nested(t)
        Item(a) -> a
      }
      + sum_nested(rest)
  }
}
```

```
// Sum a nested list of integers with list.map
pub fn sum_nested(s: NestedList(Int)) -> Int {
  int.sum(
    list.map(s, fn(x) {
      case x {
        Sublist(t) -> sum_nested(t)
        Item(a) -> a
      }
    })
  )
}
```

https://is.gd/61a_sumnested

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