Table of Contents

Moonbit/Core Result

Overview

Result[T,E] is a type used for handling computation results and errors in an exp
licit and declarative manner, similar to Rust (Result<T,E>) and OCaml (('a,
'e) result). It is an enum with two variants: Ok(T), which represents succes
s and contains a value of type T, and Err(E), representing error and containin
g an error value of type E.

Usage

Constructing Result

You can create a Result value using the Ok and Err constructors, remember to giv
e proper type annotations.

```

test {
  let _result : Result[Int, String] = Ok(42)
  let _error : Result[Int, String] = Err("Error message")

}
```

Or use the ok and err functions to create a Result value.

```

test {
  let _result : Result[String, Unit] = Ok("yes")
  let _error : Result[Int, String] = Err("error")

}
```

Querying variant

You can check the variant of a Result using the is_ok and is_err methods.

```

test {
  let result : Result[Int, String] = Ok(42)
  let is_ok = result is Ok(_)
  assert_eq(is_ok, true)
  let is_err = result is Err(_)
  assert_eq(is_err, false)
}
```

Extracting values

You can extract the value from a Result using the match expression (Pattern Mat
ching).

```

test {
  let result : Result[Int, Unit] = Ok(33)
  let val = match result {
    Ok(value) => value
    Err(_) => -1
  }
  assert_eq(val, 33)
}
```

Or using the unwrap method, which will panic if the result is Err and return the value if it is Ok.

```

test {
  let result : Result[Int, String] = Ok(42)
  let value = result.unwrap()
  assert_eq(value, 42)
}
```

A safe alternative is the or method, which returns the value if the result is Ok or a default value if it is Err.

```

test {
  let result : Result[Int, String] = Err("error")
  let value = result.or(0)
  assert_eq(value, 0)
}
```

There is a lazy version of or called or_else, which takes a function that returns a default value.

```

test {
  let result : Result[Int, String] = Err("error")
  let value = result.or_else(() => 0)
  assert_eq(value, 0)
}
```

Transforming values

To transform values inside a Result, you can use the map method, which applies a function to the value if the result is Ok, and remains unchanged if it is Err.

```

test {
  let result : Result[Int, String] = Ok(42)
  let new_result = result.map(x => x + 1)
  assert_eq(new_result, Ok(43))
}
```

A dual method to map is map_err, which applies a function to the error value if the result is Err, and remains unchanged if it is Ok.

```

test {
  let result : Result[Int, String] = Err("error")
  let new_result = result.map_err(x => x + "!")
  assert_eq(new_result, Err("error!"))
}
```

You can turn a Result[T, E] into a Option[T] by using the method to_option, which returns Some(value) if the result is Ok, and None if it is Err.

```
test {
  let result : Result[Int, String] = Ok(42)
  let option = result.to_option()
  assert_eq(option, Some(42))
  let result1 : Result[Int, String] = Err("error")
  let option1 = result1.to_option()
  assert_eq(option1, None)
}
```

Monadic operations

Moonbit provides monadic operations for Result, such as flatten and bind, which allow chaining of computations that return Result.

```
test {
  let result : Result[Result[Int, String], String] = Ok(Ok(42))
  let flattened = result.flatten()
  assert_eq(flattened, Ok(42))
}
```

The bind method is similar to map, but the function passed to it should return a Result value.

```
test {
  let result : Result[Int, String] = Ok(42)
  let new_result = result.bind(x => Ok(x + 1))
  assert_eq(new_result, Ok(43))
}
```