

Table of Contents

1	Test Package Documentation
1.1	Basic Test Structure
1.2	Assertion Functions
1.2.1	Object Identity Testing
1.2.2	Failure Testing
1.3	Test Output and Logging
1.4	Snapshot Testing
1.5	Advanced Testing Patterns
1.5.1	Testing with Complex Data
1.5.2	Error Condition Testing
1.5.3	Property-Based Testing
1.6	Test Organization
1.6.1	Grouping Related Tests
1.6.2	Setup and Teardown Patterns
1.7	Testing Best Practices
1.7.1	Clear Test Names
1.7.2	One Concept Per Test
1.7.3	Use Meaningful Test Data
1.8	Integration with MoonBit Build System
1.9	Common Testing Patterns
1.10	Performance Considerations
2	byte
2.1	Constants
2.2	Conversion
2.3	Byte Literals
3	Builtin Package Documentation
3.1	Core Types and Functions
3.1.1	Assertions and Testing
3.1.2	Inspect Function
3.2	Result Type
3.3	Option Type
3.4	Iterator Type
3.5	Array and FixedArray
3.6	String Operations
3.7	StringBuilder
3.8	JSON Support
3.9	Comparison Operations
3.10	Utility Functions
3.11	Error Handling
3.12	Best Practices
3.13	Performance Notes
4	HashMap
5	Usage
5.1	Create
5.2	Set & Get
5.3	Remove
5.4	Contains
5.5	Size & Capacity
5.6	Clear
5.7	Iteration
6	String Package Documentation
6.1	String Creation and Conversion
6.2	String Iteration
6.3	String Conversion
6.4	Unicode Handling
6.5	String Comparison
6.6	String Views
6.7	Practical Examples
6.8	Performance Notes
7	Moonbit/Core Result
7.1	Overview

Test Package Documentation

- hello
 - moonbit
 - test
- world

This package provides testing utilities and assertion functions for MoonBit programs. It includes functions for comparing values, checking object identity, and creating structured test outputs with snapshot testing capabilities.

Basic Test Structure

MoonBit tests are written using the test keyword:

```
1  ///|
2  test "basic test example" {
3      let result = 2 + 2
4      inspect(result, content="4")
5
6      // Test passes if no errors are raised
7  }
```

Assertion Functions

Object Identity Testing

Test whether two values refer to the same object in memory:

```

1  ///|
2  test "object identity" {
3    let str1 = "hello"
4    let _str2 = "hello"
5    let str3 = str1
6
7    // Same object reference
8    @test.same_object(str1, str3) // Passes - same reference
9
10   // Different objects (even if equal values)
11
12   // @test.is_not(str1, str2)
13   // May or may not pass - different string objects
14   // depend on how compiler optimization works
15   // here we interned
16
17   // Arrays and object identity
18   let arr1 = [1, 2, 3]
19   let _arr2 = [1, 2, 3]
20   let arr3 = arr1
21   @test.same_object(arr1, arr3) // Passes - same array reference @test.i
22 }

```

Failure Testing

Explicitly fail tests with custom messages:

```

1  ///|
2  test "conditional failure" {
3    let value = 10
4    if value < 0 {
5      @test.fail("Value should not be negative: \{value}")
6    }
7
8    // Test continues if condition is not met
9    inspect(value, content="10")
10 }

```

Test Output and Logging

Create structured test outputs using the Test type:

```

1  ///|
2  test "test output" {
3    let t = @test.new("Example Test")
4
5    // Write output to test buffer
6    t.write("Testing basic functionality: ")
7    t.writeln("PASS")
8
9    // Write multiple lines
10   t.writeln("Step 1: Initialize data")
11   t.writeln("Step 2: Process data")
12   t.writeln("Step 3: Verify results")
13
14   // The test output is captured for reporting
15 }

```

Snapshot Testing

Compare test outputs against saved snapshots:

```

1  ///|
2  test "snapshot testing" {
3    let t = @test.new("Snapshot Test")
4
5    // Generate some output
6    t.writeln("Current timestamp: 2024-01-01")
7    t.writeln("Processing items: [1, 2, 3, 4, 5]")
8    t.writeln("Result: SUCCESS")
9
10   // Compare against snapshot file
11   // This will create or update a snapshot file
12   t.snapshot(filename="test_output")
13 }

```

Advanced Testing Patterns

Testing with Complex Data

Test functions that work with complex data structures:

```

1  ///|
2  test "complex data testing" {
3    // Test with arrays
4    let numbers = [1, 2, 3, 4, 5]
5    let doubled = numbers.map(fn(x) { x * 2 })
6    inspect(doubled, content="[2, 4, 6, 8, 10]")
7
8    // Test with tuples (simpler than custom structs in test examples)
9    let person_data = ("Alice", 30)
10   inspect(person_data.0, content="Alice")
11   inspect(person_data.1, content="30")
12 }

```

Error Condition Testing

Test that functions properly handle error conditions:

```
1  ///|
2  test "error handling" {
3      fn safe_divide(a : Int, b : Int) -> Int? {
4          if b == 0 {
5              None
6          } else {
7              Some(a / b)
8          }
9      }
10
11     // Test normal case
12     let result = safe_divide(10, 2)
13     inspect(result, content="Some(5)")
14
15     // Test error case
16     let error_result = safe_divide(10, 0)
17     inspect(error_result, content="None")
18 }
```

Property-Based Testing

Test properties that should hold for various inputs:

```
1  ///|
2  test "property testing" {
3      fn is_even(n : Int) -> Bool {
4          n % 2 == 0
5      }
6
7      // Test the property with multiple values
8      let test_values = [0, 2, 4, 6, 8, 10]
9      for value in test_values {
10         if not(is_even(value)) {
11             @test.fail("Expected \{value} to be even")
12         }
13     }
14
15     // Test negative cases
16     let odd_values = [1, 3, 5, 7, 9]
17     for value in odd_values {
18         if is_even(value) {
19             @test.fail("Expected \{value} to be odd")
20         }
21     }
22 }
```

Test Organization

Grouping Related Tests

Use descriptive test names to group related functionality:

```
1  ///|
2  test "string operations - concatenation" {
3      let result = "hello" + " " + "world"
4      inspect(result, content="hello world")
5  }
6
7  ///|
8  test "string operations - length" {
9      let text = "MoonBit"
10     inspect(text.length(), content="7")
11 }
12
13 ///|
14 test "string operations - substring" {
15     let text = "Hello, World!"
16     let sub = text.length() // Just test length instead of substring
17     inspect(sub, content="13")
18 }
```

Setup and Teardown Patterns

Create helper functions for common test setup:

```
1  ///|
2  test "with setup helper" {
3      fn setup_test_data() -> Array[Int] {
4          [10, 20, 30, 40, 50]
5      }
6
7      fn cleanup_test_data(_data : Array[Int]) -> Unit {
8          // Cleanup logic here
9      }
10
11     let data = setup_test_data()
12
13     // Perform tests
14     inspect(data.length(), content="5")
15     inspect(data[0], content="10")
16     inspect(data[4], content="50")
17     cleanup_test_data(data)
18 }
```

Testing Best Practices

Clear Test Names

Use descriptive names that explain what is being tested:

```

1  ///|
2  test "user_can_login_with_valid_credentials" {
3    // Test implementation
4  }
5
6  ///|
7  test "login_fails_with_invalid_password" {
8    // Test implementation
9  }
10
11 ///|
12 test "shopping_cart_calculates_total_correctly" {
13   // Test implementation
14 }

```

One Concept Per Test

Keep tests focused on a single concept:

```

1  ///|
2  /// Good - tests one specific behavior
3  test "array_push_increases_length" {
4    let arr = Array::new()
5    let initial_length = arr.length()
6    arr.push(42)
7    let new_length = arr.length()
8    inspect(new_length, content="\{initial_length + 1}")
9  }
10
11 ///|
12 /// Good - tests another specific behavior
13 test "array_push_adds_element_at_end" {
14   let arr = Array::new()
15   arr.push(10)
16   arr.push(20)
17   inspect(arr[arr.length() - 1], content="20")
18 }

```

Use Meaningful Test Data

Choose test data that makes the test's intent clear:

```

1  ///|
2  test "tax_calculation_for_standard_rate" {
3    let price = 100
4    let tax_rate = 8 // 8% tax as integer percentage
5    let calculated_tax = price * tax_rate / 100
6    inspect(calculated_tax, content="8")
7  }

```

Integration with MoonBit Build System

Tests are automatically discovered and run by the MoonBit build system:

- Use moon test to run all tests
- Use moon test --update to update snapshots
- Tests in *_test.mbt files are blackbox tests
- Tests in regular .mbt files are whitebox tests

Common Testing Patterns

- **Arrange-Act-Assert:** Set up data, perform operation, verify result
- **Given-When-Then:** Given some context, when an action occurs, then verify outcome
- **Red-Green-Refactor:** Write failing test, make it pass, improve code
- **Test-Driven Development:** Write tests before implementation

Performance Considerations

- Keep tests fast by avoiding expensive operations when possible
- Use setup/teardown functions to share expensive initialization
- Consider using smaller datasets for unit tests
- Save integration tests with large datasets for separate test suites

The test package provides essential tools for ensuring code quality and correctness in MoonBit applications through comprehensive testing capabilities.

byte

A package for working with bytes (8-bit unsigned integers) in MoonBit.

Constants

The package provides constants for the minimum and maximum values of a byte:

```

1  ///|
2  test "byte constants" {
3      inspect(@byte.min_value, content="b'\\x00'")
4      inspect(@byte.max_value, content="b'\\xFF'")
5  }
```

Conversion

Bytes can be converted to other numeric types. The package provides conversion to UInt64:


```

1  ///|
2  test "byte conversion" {
3      let byte = b'A'
4      inspect(byte.to_uint64(), content="65")
5      let byte = b' '
6      inspect(byte.to_uint64(), content="32")
7  }

```

Byte Literals

Although not directly part of this package, MoonBit provides byte literals with the `b` prefix:

```

1  ///|
2  test "byte literals" {
3      // ASCII character
4      let a = b'a'
5      inspect(a.to_uint64(), content="97")
6
7      // Hexadecimal escape sequence
8      let hex = b'\x41'
9      inspect(hex.to_uint64(), content="65")
10
11     // Null byte
12     let null = b'\x00'
13     inspect(null.to_uint64(), content="0")
14
15     // Maximum value
16     let max = b'\xff'
17     inspect(max.to_uint64(), content="255")
18 }

```

Note: The same conversion method can be called either as a method (

```

1  ///|
2  test "assertions" {
3    // Basic equality assertion
4    assert_eq(1 + 1, 2)
5    assert_eq("hello", "hello")
6
7    // Boolean assertions
8    assert_true(5 > 3)
9    assert_false(2 > 5)
10
11   // Inequality assertion
12   assert_not_eq(1, 2)
13   assert_not_eq("foo", "bar")
14 }

```

Inspect Function

The inspect function is used for testing and debugging:

```

1  ///|
2  test "inspect usage" {
3    let value = 42
4    inspect(value, content="42")
5    let list = [1, 2, 3]
6    inspect(list, content="[1, 2, 3]")
7    let result : Result[Int, String] = Ok(100)
8    inspect(result, content="Ok(100)")
9  }

```

Result Type

The Result[T, E] type represents operations that can succeed or fail:

```

1  ///|
2  test "result type" {
3      fn divide(a : Int, b : Int) -> Result[Int, String] {
4          if b == 0 {
5              Err("Division by zero")
6          } else {
7              Ok(a / b)
8          }
9      }
10
11     // Success case
12     let result1 = divide(10, 2)
13     inspect(result1, content="Ok(5)")
14
15     // Error case
16     let result2 = divide(10, 0)
17     inspect(result2, content="Err(\"Division by zero\")")
18
19     // Pattern matching on Result
20     match result1 {
21         Ok(value) => inspect(value, content="5")
22         Err(_) => inspect(false, content="true")
23     }
24 }

```

Option Type

The Option[T] type represents values that may or may not exist:

```

1  ///|
2  test "option type" {
3      fn find_first_even(numbers : Array[Int]) -> Int? {
4          for num in numbers {
5              if num % 2 == 0 {
6                  return Some(num)
7              }
8          }
9          None
10     }
11
12     // Found case
13     let result1 = find_first_even([1, 3, 4, 5])
14     inspect(result1, content="Some(4)")
15
16     // Not found case
17     let result2 = find_first_even([1, 3, 5])
18     inspect(result2, content="None")
19
20     // Pattern matching on Option
21     match result1 {
22         Some(value) => inspect(value, content="4")
23         None => inspect(false, content="true")
24     }
25 }

```

Iterator Type

The `Iter[T]` type provides lazy iteration over sequences:

```
1  ///|
2  test "iterators" {
3    // Create iterator from array
4    let numbers = [1, 2, 3, 4, 5]
5    let iter = numbers.iter()
6
7    // Collect back to array
8    let collected = iter.collect()
9    inspect(collected, content="[1, 2, 3, 4, 5]")
10
11   // Map transformation
12   let doubled = numbers.iter().map(fn(x) { x * 2 }).collect()
13   inspect(doubled, content="[2, 4, 6, 8, 10]")
14
15   // Filter elements
16   let evens = numbers.iter().filter(fn(x) { x % 2 == 0 }).collect()
17   inspect(evens, content="[2, 4]")
18
19   // Fold (reduce) operation
20   let sum = numbers.iter().fold(init=0, fn(acc, x) { acc + x })
21   inspect(sum, content="15")
22 }
```

Array and FixedArray

Built-in array types for storing collections:

```
1  ///|
2  test "arrays" {
3    // Dynamic arrays
4    let arr = Array::new()
5    arr.push(1)
6    arr.push(2)
7    arr.push(3)
8    inspect(arr, content="[1, 2, 3]")
9
10   // Array from literal
11   let fixed_arr = [10, 20, 30]
12   inspect(fixed_arr, content="[10, 20, 30]")
13
14   // Array operations
15   let length = fixed_arr.length()
16   inspect(length, content="3")
17   let first = fixed_arr[0]
18   inspect(first, content="10")
19 }
```

String Operations

Basic string functionality:

```
1  ///|
2  test "strings" {
3    let text = "Hello, World!"
4
5    // String length
6    let len = text.length()
7    inspect(len, content="13")
8
9    // String concatenation
10   let greeting = "Hello" + ", " + "World!"
11   inspect(greeting, content="Hello, World!")
12
13   // String comparison
14   let equal = "test" == "test"
15   inspect(equal, content="true")
16 }
```

StringBuilder

Efficient string building:

```
1  ///|
2  test "string builder" {
3    let builder = StringBuilder::new()
4    builder.write_string("Hello")
5    builder.write_string(", ")
6    builder.write_string("World!")
7    let result = builder.to_string()
8    inspect(result, content="Hello, World!")
9  }
```

JSON Support

Basic JSON operations:

```

1  ///|
2  test "json" {
3    // JSON values
4    let json_null = null
5    inspect(json_null, content="Null")
6    let json_bool = true.to_json()
7    inspect(json_bool, content="True")
8    let json_number = (42 : Int).to_json()
9    inspect(json_number, content="Number(42)")
10   let json_string = "hello".to_json()
11   inspect(
12     json_string,
13     content=(
14       #|String("hello")
15     ),
16   )
17 }

```

Comparison Operations

Built-in comparison operators:

```

1  ///|
2  test "comparisons" {
3    // Equality
4    inspect(5 == 5, content="true")
5    inspect(5 != 3, content="true")
6
7    // Ordering
8    inspect(3 < 5, content="true")
9    inspect(5 > 3, content="true")
10   inspect(5 >= 5, content="true")
11   inspect(3 <= 5, content="true")
12
13   // String comparison
14   inspect("apple" < "banana", content="true")
15   inspect("hello" == "hello", content="true")
16 }

```

Utility Functions

Helpful utility functions:

```

1  ///|
2  test "utilities" {
3    // Identity and ignore
4    let value = 42
5    ignore(value) // Discards the value
6
7    // Boolean negation
8    let result = not(false)
9    inspect(result, content="true")
10
11   // Physical equality (reference equality)
12   let arr1 = [1, 2, 3]
13   let arr2 = [1, 2, 3]
14   let same_ref = arr1
15   inspect(physical_equal(arr1, arr2), content="false") // Different objects
16   inspect(physical_equal(arr1, same_ref), content="true") // Same reference
17 }

```

Error Handling

Basic error handling with panic and abort:

```

1  ///|
2  test "error handling" {
3    // This would panic in a real scenario, but we demonstrate the concept
4    fn safe_divide(a : Int, b : Int) -> Int {
5      if b == 0 {
6        // In real code: panic()
7        // For testing, we return a default value
8        0
9      } else {
10       a / b
11     }
12   }
13
14   let result = safe_divide(10, 2)
15   inspect(result, content="5")
16   let safe_result = safe_divide(10, 0)
17   inspect(safe_result, content="0")
18 }

```

Best Practices

- **Use assertions liberally in tests:** They help catch bugs early and document expected behavior
- **Prefer Result over exceptions:** For recoverable errors, use `Result[T, E]` instead of panicking
- **Use Option for nullable values:** Instead of null pointers, use `Option[T]`
- **Leverage iterators for data processing:** They provide composable and efficient data transformations
- **Use StringBuilder for string concatenation:** More efficient than repeated string concatenation
- **Pattern match on Result and Option:** Handle both success and failure cases explicitly

Performance Notes

- Arrays have $O(1)$ access and $O(1)$ amortized append
- Iterators are lazy and don't allocate intermediate collections
- `StringBuilder` is more efficient than string concatenation for building large strings
- Physical equality is faster than structural equality but should be used carefully

HashMap

A mutable hash map based on a Robin Hood hash table.

Usage

Create

You can create an empty map using `new()` or construct it using `from_array()`.

```
1  /// |
2  test {
3      let _map2 : @hashmap.HashMap[String, Int] = @hashmap.new()
4
5  }
```

Set & Get

You can use `set()` to add a key-value pair to the map, and use `get()` to get a value.


```

1  ///|
2  test {
3      let map : @hashmap.HashMap[String, Int] = @hashmap.new()
4      map.set("a", 1)
5      assert_eq(map.get("a"), Some(1))
6      assert_eq(map.get_or_default("a", 0), 1)
7      assert_eq(map.get_or_default("b", 0), 0)
8      map.remove("a")
9      assert_eq(map.contains("a"), false)
10 }

```

Remove

You can use `remove()` to remove a key-value pair.

```

1  ///|
2  test {
3      let map = @hashmap.of([("a", 1), ("b", 2), ("c", 3)])
4      map.remove("a") |> ignore
5      assert_eq(map.to_array(), [("c", 3), ("b", 2)])
6  }

```

Contains

You can use `contains()` to check whether a key exists.

```

1  ///|
2  test {
3      let map = @hashmap.of([("a", 1), ("b", 2), ("c", 3)])
4      assert_eq(map.contains("a"), true)
5      assert_eq(map.contains("d"), false)
6  }

```

Size & Capacity

You can use `size()` to get the number of key-value pairs in the map, or `capacity()` to get the current capacity.

```

1  ///|
2  test {
3      let map = @hashmap.of([("a", 1), ("b", 2), ("c", 3)])
4      assert_eq(map.size(), 3)
5      assert_eq(map.capacity(), 8)
6  }

```

Similarly, you can use `is_empty()` to check whether the map is empty.

```

1  ///|
2  test {
3      let map : @hashmap.HashMap[String, Int] = @hashmap.new()
4      assert_eq(map.is_empty(), true)
5  }

```

Clear

You can use `clear` to remove all key-value pairs from the map, but the allocated memory will not change.

```
1  ///|
2  test {
3      let map = @hashmap.of([("a", 1), ("b", 2), ("c", 3)])
4      map.clear()
5      assert_eq(map.is_empty(), true)
6  }
```

Iteration

You can use `each()` or `eachi()` to iterate through all key-value pairs.

```
1  ///|
2  test {
3      let map = @hashmap.of([("a", 1), ("b", 2), ("c", 3)])
4      let arr = []
5      map.each((k, v) => arr.push((k, v)))
6      let arr2 = []
7      map.eachi((i, k, v) => arr2.push((i, k, v)))
8  }
```

Or use `iter()` to get an iterator of hashmap.

```
1  ///|
2  test {
3      let map = @hashmap.of([("a", 1), ("b", 2), ("c", 3)])
4      let _iter = map.iter()
5
6  }
```

String Package Documentation

This package provides comprehensive string manipulation utilities for MoonBit, including string creation, conversion, searching, and Unicode handling.

String Creation and Conversion

Create strings from various sources:

```

1  ///|
2  test "string creation" {
3    // From character array
4    let chars = ['H', 'e', 'l', 'l', 'o']
5    let str1 = String::from_array(chars)
6    inspect(str1, content="Hello")
7
8    // From character iterator
9    let str2 = String::from_iter(['W', 'o', 'r', 'l', 'd'].iter())
10   inspect(str2, content="World")
11
12   // Default empty string
13   let empty = String::default()
14   inspect(empty, content="")
15 }

```

String Iteration

Iterate over Unicode characters in strings:

```

1  ///|
2  test "string iteration" {
3    let text = "Hello??"
4
5    // Forward iteration
6    let chars = text.iter().collect()
7    inspect(chars, content="['H', 'e', 'l', 'l', 'o', '??']")
8
9    // Reverse iteration
10   let reversed = text.rev_iter().collect()
11   inspect(reversed, content="['??', 'o', 'l', 'l', 'e', 'H']")
12
13   // Iteration with indices - demonstrate iter2 functionality
14   let mut count = 0
15   let mut first_char = 'a'
16   text
17   .iter2()
18   .each(fn(idx, char) {
19     if idx == 0 {
20       first_char = char
21     }
22     count = count + 1
23   })
24   inspect(first_char, content="H")
25   inspect(count, content="6") // 6 Unicode characters
26 }

```

String Conversion

Convert strings to other formats:

```

1  ///|
2  test "string conversion" {
3    let text = "Hello"
4
5    // Convert to character array
6    let chars = text.to_array()
7    inspect(chars, content=["H', 'e', 'l', 'l', 'o'])
8
9    // Convert to bytes (UTF-16 LE encoding)
10   let bytes = text.to_bytes()
11   inspect(bytes.length(), content="10") // 5 chars * 2 bytes each
12 }

```

Unicode Handling

Work with Unicode characters and surrogate pairs:

```

1  ///|
2  test "unicode handling" {
3    let emoji_text = "Hello??World"
4
5    // Character count vs UTF-16 code unit count
6    let char_count = emoji_text.iter().count()
7    let code_unit_count = emoji_text.length()
8    inspect(char_count, content="11") // Unicode characters
9    inspect(code_unit_count, content="12") // UTF-16 code units
10
11   // Find character offset
12   let offset = emoji_text.offset_of_nth_char(5) // Position of emoji
13   inspect(offset, content="Some(5)")
14
15   // Test character length
16   let has_11_chars = emoji_text.char_length_eq(11)
17   inspect(has_11_chars, content="true")
18 }

```

String Comparison

Strings are ordered using shortlex order by Unicode code points:

```

1  ///|
2  test "string comparison" {
3    let result1 = "apple".compare("banana")
4    inspect(result1, content="-1") // apple < banana
5    let result2 = "hello".compare("hello")
6    inspect(result2, content="0") // equal
7    let result3 = "zebra".compare("apple")
8    inspect(result3, content="1") // zebra > apple
9  }

```

String Views

String views provide efficient substring operations without copying:

```

1  ///|
2  test "string views" {
3      let text = "Hello, World!"
4      let view = text[:][7:12] // "World" - create view using slice notation
5
6      // Views support similar operations as strings
7      let chars = view.iter().collect()
8      inspect(chars, content="['W', 'o', 'r', 'l', 'd']")
9
10     // Convert view back to string
11     let substring = view.to_string()
12     inspect(substring, content="World")
13 }

```

Practical Examples

Common string manipulation tasks:

```

1  ///|
2  test "practical examples" {
3      let text = "The quick brown fox"
4
5      // Split into words (using whitespace) - returns Iter[View]
6      let words = text.split(" ").collect()
7      inspect(words.length(), content="4")
8      inspect(words[0].to_string(), content="The")
9      inspect(words[3].to_string(), content="fox")
10
11     // Join words back together - convert views to strings first
12     let word_strings = words.map(fn(v) { v.to_string() })
13     let mut result = ""
14     for i, word in word_strings.iter2() {
15         if i > 0 {
16             result = result + "-"
17         }
18         result = result + word
19     }
20     inspect(result, content="The-quick-brown-fox")
21
22     // Case conversion (works on views)
23     let upper = text[:].to_upper().to_string()
24     inspect(upper, content="THE QUICK BROWN FOX")
25     let lower = text[:].to_lower().to_string()
26     inspect(lower, content="the quick brown fox")
27 }

```

Performance Notes

- Use `StringBuilder` or `Buffer` for building strings incrementally rather than repeated concatenation
- String views are lightweight and don't copy the underlying data
- Unicode iteration handles surrogate pairs correctly but is slower than UTF-16 code unit iteration
- Character length operations (`char_length_eq`, `char_length_ge`) have $O(n)$ complexity where n is the character count

Moonbit/Core Result

Overview

`Result[T,E]` is a type used for handling computation results and errors in an explicit 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 success and contains a value of type `T`, and `Err(E)`, representing error and containing an error value of type `E`.

Usage

Constructing Result

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

```
1  ///|
2  test {
3      let _result : Result[Int, String] = Ok(42)
4      let _error : Result[Int, String] = Err("Error message")
5
6  }
```

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

```
1  ///|
2  test {
3      let _result : Result[String, Unit] = Ok("yes")
4      let _error : Result[Int, String] = Err("error")
5
6  }
```

Querying variant

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

```
1  ///|
2  test {
3      let result : Result[Int, String] = Ok(42)
4      let is_ok = result is Ok(_)
5      assert_eq(is_ok, true)
6      let is_err = result is Err(_)
7      assert_eq(is_err, false)
8  }
```

Extracting values

You can extract the value from a `Result` using the `match` expression (Pattern Matching).

```
1  ///|
2  test {
3      let result : Result[Int, Unit] = Ok(33)
4      let val = match result {
5          Ok(value) => value
6          Err(_)   => -1
7      }
8      assert_eq(val, 33)
9  }
```

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

```
1  ///|
2  test {
3      let result : Result[Int, String] = Ok(42)
4      let value = result.unwrap()
5      assert_eq(value, 42)
6  }
```

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

```
1  ///|
2  test {
3      let result : Result[Int, String] = Err("error")
4      let value = result.or(0)
5      assert_eq(value, 0)
6  }
```

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

```
1  ///|
2  test {
3      let result : Result[Int, String] = Err("error")
4      let value = result.or_else(() => 0)
5      assert_eq(value, 0)
6  }
```

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`.

```
1  ///|
2  test {
3      let result : Result[Int, String] = Ok(42)
4      let new_result = result.map(x => x + 1)
5      assert_eq(new_result, Ok(43))
6  }
```

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`.

```

1  ///|
2  test {
3      let result : Result[Int, String] = Err("error")
4      let new_result = result.map_err(x => x + "!")
5      assert_eq(new_result, Err("error!"))
6  }

```

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`.

```

1  ///|
2  test {
3      let result : Result[Int, String] = Ok(42)
4      let option = result.to_option()
5      assert_eq(option, Some(42))
6      let result1 : Result[Int, String] = Err("error")
7      let option1 = result1.to_option()
8      assert_eq(option1, None)
9  }

```

Monadic operations

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

```

1  ///|
2  test {
3      let result : Result[Result[Int, String], String] = Ok(Ok(42))
4      let flattened = result.flatten()
5      assert_eq(flattened, Ok(42))
6  }

```

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

```

1  ///|
2  test {
3      let result : Result[Int, String] = Ok(42)
4      let new_result = result.bind(x => Ok(x + 1))
5      assert_eq(new_result, Ok(43))
6  }

```

uint16

The `moonbitlang/core/uint16` package provides functionality for working with 16-bit unsigned integers. This package includes constants, operators, and conversions for `UInt16` values.

Constants

The package defines the minimum and maximum values for `UInt16`:


```

1  ///|
2  test "UInt16 constants" {
3      // Minimum value of UInt16
4      inspect(@uint16.min_value, content="0")
5
6      // Maximum value of UInt16
7      inspect(@uint16.max_value, content="65535")
8  }

```

Arithmetic Operations

UInt16 supports standard arithmetic operations:

```

1  ///|
2  test "UInt16 arithmetic" {
3      let a : UInt16 = 100
4      let b : UInt16 = 50
5
6      // Addition
7      inspect(a + b, content="150")
8
9      // Subtraction
10     inspect(a - b, content="50")
11
12     // Multiplication
13     inspect(a * b, content="5000")
14
15     // Division
16     inspect(a / b, content="2")
17
18     // Overflow behavior
19     inspect(@uint16.max_value + 1, content="0") // Wraps around to 0
20     inspect(@uint16.min_value - 1, content="65535") // Underflow wraps to
21 }

```

Bitwise Operations

UInt16 supports various bitwise operations:

```

1  ///|
2  test "UInt16 bitwise operations" {
3    let a : UInt16 = 0b1010
4    let b : UInt16 = 0b1100
5
6    // Bitwise AND
7    inspect(a & b, content="8")
8
9    // Bitwise OR
10   inspect(a | b, content="14")
11
12   // Bitwise XOR
13   inspect(a ^ b, content="6")
14
15   // Left shift
16   inspect(a << 1, content="20")
17   inspect(a << 2, content="40")
18
19   // Right shift
20   inspect(a >> 1, content="5")
21   inspect(b >> 2, content="3")
22 }

```

Comparison and Equality

UInt16 supports comparison and equality operations:

```

1  ///|
2  test "UInt16 comparison and equality" {
3    let a : UInt16 = 100
4    let b : UInt16 = 50
5    let c : UInt16 = 100
6
7    // Equality
8    inspect(a == c, content="true")
9    inspect(a != b, content="true")
10
11   // Comparison
12   inspect(a > b, content="true")
13   inspect(b < a, content="true")
14   inspect(a >= c, content="true")
15   inspect(c <= a, content="true")
16 }

```

Default Value and Hashing

UInt16 implements the Default trait:

```

1  ///|
2  test "UInt16 default value" {
3    // Default value is 0
4    let a : UInt16 = 0
5    inspect(a, content="0")
6
7    // Hash support is available via .hash()
8    let value : UInt16 = 42
9    inspect(value.hash(), content="42")
10 }

```

Type Conversions

UInt16 works with various conversions to and from other types:

```

1  ///|
2  test "UInt16 conversions" {
3    // From Int to UInt16
4    inspect((42).to_uint16(), content="42")
5
6    // From UInt16 to Int
7    let value : UInt16 = 100
8    inspect(value.to_int(), content="100")
9
10   // Overflow handling in conversions
11   inspect((-1).to_uint16(), content="65535") // Negative numbers wrap around
12   inspect((65536).to_uint16(), content="0") // Values too large wrap around
13   inspect((65537).to_uint16(), content="1") // 65536 + 1 = 1 (modulo 65536)
14
15   // From Byte to UInt16
16   inspect(b'A'.to_uint16(), content="65")
17   inspect(b'\xFF'.to_uint16(), content="255")
18 }

```

Tuple

Tuple is a fixed-size collection of elements of different types. It is a lightweight data structure that can be used to store multiple values in a single variable. This sub-package introduces utils for binary tuples.

Usage

Create

Create a new tuple using the tuple literal syntax.

```

1  ///|
2  test {
3      let tuple2 = (1, 2)
4      let tuple3 = (1, 2, 3)
5      inspect((tuple2, tuple3), content="((1, 2), (1, 2, 3))")
6  }

```

Access

You can access the elements of the tuple using pattern match or dot access.

```

1  ///|
2  test {
3      let tuple = (1, 2)
4      assert_eq(tuple.0, 1)
5      assert_eq(tuple.1, 2)
6      let (a, b) = tuple
7      assert_eq(a, 1)
8      assert_eq(b, 2)
9  }

```

Transformation

You can transform the tuple using the matrix functions combined with then.

```

1  ///|
2  test {
3      let tuple = (1, 2)
4      let tuple2 = ((pair : (Int, Int)) => (pair.0 + 1, pair.1))(tuple)
5      inspect(tuple2, content="(2, 2)")
6      let tuple3 = tuple |> then(pair => (pair.0, pair.1 + 1))
7      inspect(tuple3, content="(1, 3)")
8      let mapped = tuple |> then(pair => (pair.0 + 1, pair.1 - 1))
9      inspect(mapped, content="(2, 1)")
10 }

```

List

The List package provides an immutable linked list data structure with a variety of utility functions for functional programming.

Table of Contents

- Overview
- Performance
- Usage
 - Create
 - Basic Operations
 - Access Elements
 - Iteration
 - Advanced Operations
 - Conversion
 - Equality
- Error Handling Best Practices
- Implementation Notes
- Comparison with Other Collections

Overview

List is a functional, immutable data structure that supports efficient traversal, transformation, and manipulation. It is particularly useful for recursive algorithms and scenarios where immutability is required.

Performance

- **prepend**: $O(1)$
- **length**: $O(n)$
- **map/filter**: $O(n)$
- **concatenate**: $O(n)$
- **reverse**: $O(n)$
- **nth**: $O(n)$
- **sort**: $O(n \log n)$
- **flatten**: $O(n * m)$, where m is the average inner list length
- **space complexity**: $O(n)$

Usage

Create

You can create an empty list or a list from an array.

```

1  ///|
2  test {
3      let empty_list : @list.List[Int] = @list.new()
4      assert_true(empty_list.is_empty())
5      let list = @list.of([1, 2, 3, 4, 5])
6      assert_eq(list, @list.of([1, 2, 3, 4, 5]))
7  }

```

Basic Operations

Prepend

Add an element to the beginning of the list.

```

1  ///|
2  test {
3      let list = @list.of([2, 3, 4, 5]).prepend(1)
4      assert_eq(list, @list.of([1, 2, 3, 4, 5]))
5  }

```

Length

Get the number of elements in the list.

```

1  ///|
2  test {
3      let list = @list.of([1, 2, 3, 4, 5])
4      assert_eq(list.length(), 5)
5  }

```

Check if Empty

Determine if the list is empty.

```

1  ///|
2  test {
3      let empty_list : @list.List[Int] = @list.new()
4      assert_eq(empty_list.is_empty(), true)
5  }

```

Access Elements

Head

Get the first element of the list as an Option.

```

1  ///|
2  test {
3    let list = @list.of([1, 2, 3, 4, 5])
4    assert_eq(list.head(), Some(1))
5  }

```

Tail

Get the list without its first element.

```

1  ///|
2  test {
3    let list = @list.of([1, 2, 3, 4, 5])
4    assert_eq(list.unsafe_tail(), @list.of([2, 3, 4, 5]))
5  }

```

Nth Element

Get the nth element of the list as an Option.

```

1  ///|
2  test {
3    let list = @list.of([1, 2, 3, 4, 5])
4    assert_eq(list.nth(2), Some(3))
5  }

```

Iteration

Each

Iterate over the elements of the list.

```

1  ///|
2  test {
3    let arr = []
4    @list.of([1, 2, 3, 4, 5]).each(x => arr.push(x))
5    assert_eq(arr, [1, 2, 3, 4, 5])
6  }

```

Map

Transform each element of the list.

```

1  ///|
2  test {
3    let list = @list.of([1, 2, 3, 4, 5]).map(x => x * 2)
4    assert_eq(list, @list.of([2, 4, 6, 8, 10]))
5  }

```

Filter

Keep elements that satisfy a predicate.

```

1  ///|
2  test {
3      let list = @list.of([1, 2, 3, 4, 5]).filter(x => x % 2 == 0)
4      assert_eq(list, @list.of([2, 4]))
5  }

```

Advanced Operations

Reverse

Reverse the list.

```

1  ///|
2  test {
3      let list = @list.of([1, 2, 3, 4, 5]).rev()
4      assert_eq(list, @list.of([5, 4, 3, 2, 1]))
5  }

```

Concatenate

Concatenate two lists.

```

1  ///|
2  test {
3      let list = @list.of([1, 2, 3]).concat(@list.of([4, 5]))
4      assert_eq(list, @list.of([1, 2, 3, 4, 5]))
5  }

```

Flatten

Flatten a list of lists.

```

1  ///|
2  test {
3      let list = @list.of([@list.of([1, 2]), @list.of([3, 4])]).flatten()
4      assert_eq(list, @list.of([1, 2, 3, 4]))
5  }

```

Sort

Sort the list in ascending order.

```

1  ///|
2  test {
3      let list = @list.of([3, 1, 4, 1, 5, 9]).sort()
4      assert_eq(list, @list.of([1, 1, 3, 4, 5, 9]))
5  }

```

Conversion

To Array

Convert a list to an array.

```
1  ///|
2  test {
3      let list = @list.of([1, 2, 3, 4, 5])
4      assert_eq(list.to_array(), [1, 2, 3, 4, 5])
5  }
```

From Array

Create a list from an array.

```
1  ///|
2  test {
3      let list = @list.from_array([1, 2, 3, 4, 5])
4      assert_eq(list, @list.of([1, 2, 3, 4, 5]))
5  }
```

Equality

Lists with the same elements in the same order are considered equal.

```
1  ///|
2  test {
3      let list1 = @list.of([1, 2, 3])
4      let list2 = @list.of([1, 2, 3])
5      assert_eq(list1 == list2, true)
6  }
```

Error Handling Best Practices

When accessing elements that might not exist, use pattern matching for safety:

```

1  ///|
2  fn safe_head(list : @list.List[Int]) -> Int {
3      match list.head() {
4          Some(value) => value
5          None => 0 // Default value
6      }
7  }
8
9  ///|
10 test {
11     let list = @list.of([1, 2, 3])
12     assert_eq(safe_head(list), 1)
13     let empty_list : @list.List[Int] = @list.new()
14     assert_eq(safe_head(empty_list), 0)
15 }

```

Additional Error Cases

- **nth()** on an empty list or out-of-bounds index: Returns None.
- **tail()** on an empty list: Returns Empty.
- **sort()** with non-comparable elements: Throws a runtime error.

Implementation Notes

The List is implemented as a singly linked list. Operations like prepend and head are O(1), while operations like length and map are O(n).

Key properties of the implementation:

- Immutable by design
- Recursive-friendly
- Optimized for functional programming patterns

Comparison with Other Collections

- **@array.T**: Provides O(1) random access but is mutable; use when random access is required.
- **@list.T**: Immutable and optimized for recursive operations; use when immutability and functional patterns are required.

Choose List when you need:

- Immutable data structures
- Efficient prepend operations
- Functional programming patterns

Priority Queue

A priority queue is a data structure capable of maintaining maximum/minimum values at front of the queue, which may have other names in other programming languages (C++ `std::priority_queue` / Rust `BinaryHeap`). The priority queue here is implemented as a pairing heap and has excellent performance.

Usage

Create

You can use `new()` or `of()` to create a priority queue.

```
1  ///|
2  test {
3      let queue1 : @priority_queue.T[Int] = @priority_queue.new()
4      let queue2 = @priority_queue.of([1, 2, 3])
5      @json.inspect(queue1, content=[])
6      @json.inspect(queue2, content=[3, 2, 1])
7  }
```

Note, however, that the default priority queue created is greater-first; if you need to create a less-first queue, you can write a struct belongs to `Compare` trait to implement it.

Length

You can use `length()` to get the number of elements in the current priority queue.

```
1  ///|
2  test {
3      let pq = @priority_queue.of([1, 2, 3, 4, 5])
4      assert_eq(pq.length(), 5)
5  }
```

Similarly, you can use the `is_empty` to determine whether the priority queue is empty.

```
1  ///|
2  test {
3      let pq : @priority_queue.T[Int] = @priority_queue.new()
4      assert_eq(pq.is_empty(), true)
5  }
```

Peek

You can use `peek()` to look at the head element of a queue, which must be either the maximum or minimum value of an element in the queue, depending on the nature of the specification. The return value of `peek()` is an `Option`, which means that the result will be `None` when the queue is empty.

```

1  ///|
2  test {
3      let pq = @priority_queue.of([1, 2, 3, 4, 5])
4      assert_eq(pq.peek(), Some(5))
5  }

```

Push

You can use `push()` to add elements to the priority queue.

```

1  ///|
2  test {
3      let pq : @priority_queue.T[Int] = @priority_queue.new()
4      pq.push(1)
5      pq.push(2)
6      assert_eq(pq.peek(), Some(2))
7  }

```

Pop

You can use `pop()` to pop the element at the front of the priority queue, respectively, and like `Peek`, its return values are `Option`, loaded with the value of the element being popped.

```

1  ///|
2  test {
3      let pq = @priority_queue.of([5, 4, 3, 2, 1])
4      assert_eq(pq.pop(), Some(5))
5  }

```

```

1  ///|
2  test {
3      let pq = @priority_queue.of([5, 4, 3, 2, 1])
4      assert_eq(pq.length(), 5)
5  }

```

Clear

You can use `clear` to clear a priority queue.

```

1  ///|
2  test {
3      let pq = @priority_queue.of([1, 2, 3, 4, 5])
4      pq.clear()
5      assert_eq(pq.is_empty(), true)
6  }

```

Copy and Transfer

You can copy a priority queue using the `copy` method.

```

1  ///|
2  test {
3    let pq = @priority_queue.of([1, 2, 3])
4    let _pq2 = pq.copy()
5
6  }

```

Array Package Documentation

This package provides array manipulation utilities for MoonBit, including fixed-size arrays (FixedArray), dynamic arrays (Array), and array views (ArrayView/View).

Creating Arrays

There are several ways to create arrays in MoonBit:

```

1  ///|
2  test "array creation" {
3    // Using array literal
4    let arr1 = [1, 2, 3]
5    inspect(arr1, content="[1, 2, 3]")
6
7    // Creating with indices
8    let arr2 = Array::makei(3, i => i * 2)
9    inspect(arr2, content="[0, 2, 4]")
10
11   // Creating from iterator
12   let arr3 = Array::from_iter("hello".iter())
13   inspect(arr3, content="['h', 'e', 'l', 'l', 'o']")
14 }

```

Array Operations

Common array operations include mapping, filtering, and folding:

```

1  ///|
2  test "array operations" {
3    let nums = [1, 2, 3, 4, 5]
4
5    // Filtering out odd numbers and negating the remaining
6    let neg_evens = nums.filter_map(x => if x % 2 == 0 { Some(-x) } else {
7    inspect(neg_evens, content="[-2, -4]")
8
9    // Summing array
10   let sum = nums.fold(init=0, (acc, x) => acc + x)
11   inspect(sum, content="15")
12
13   // Finding last element
14   let last = nums.last()
15   inspect(last, content="Some(5)")
16 }

```

Sorting

The package provides various sorting utilities:

```
1  ///|
2  test "sorting" {
3    let arr = [3, 1, 4, 1, 5, 9, 2, 6]
4
5    // Basic sorting - creates new sorted array
6    let sorted1 = arr.copy()
7    sorted1.sort()
8    inspect(sorted1, content="[1, 1, 2, 3, 4, 5, 6, 9]")
9
10   // Custom comparison
11   let strs = ["aa", "b", "ccc"]
12   let sorted2 = strs.copy()
13   sorted2.sort_by((a, b) => a.length().compare(b.length()))
14   inspect(
15     sorted2,
16     content=(
17       #|["b", "aa", "ccc"]
18     ),
19   )
20
21   // Sort by key
22   let pairs = [(2, "b"), (1, "a"), (3, "c")]
23   let sorted3 = pairs.copy()
24   sorted3.sort_by_key(p => p.0)
25   inspect(
26     sorted3,
27     content=(
28       #|[(1, "a"), (2, "b"), (3, "c")]
29     ),
30   )
31 }
```

Array Views

Array views provide a lightweight way to work with array slices:

```
1  ///|
2  test "array views" {
3    let arr = [1, 2, 3, 4, 5]
4    let view = arr[1:4]
5    inspect(view, content="[2, 3, 4]")
6
7    // Map view to new array
8    let doubled = view.map(x => x * 2)
9    inspect(doubled, content="[4, 6, 8]")
10 }
```

Fixed Arrays

Fixed arrays provide immutable array operations:

```
1  ///|
2  test "fixed arrays" {
3    let fixed : FixedArray[_] = [1, 2, 3]
4
5    // Concatenation creates new array
6    let combined = fixed + [4, 5]
7    inspect(combined, content="[1, 2, 3, 4, 5]")
8
9    // Check for containment
10   let has_two = fixed.contains(2)
11   inspect(has_two, content="true")
12
13   // Check if array starts/ends with sequence
14   let starts = fixed.starts_with([1, 2])
15   inspect(starts, content="true")
16   let ends = fixed.ends_with([2, 3])
17   inspect(ends, content="true")
18 }
```

Utilities

Additional array utilities for common operations:

```
1  ///|
2  test "utilities" {
3    // Join string array
4    let words = ["hello", "world"]
5    let joined = words.join(" ")
6    inspect(joined, content="hello world")
7
8    // Random shuffling
9    let nums = [1, 2, 3, 4, 5]
10   // Using deterministic 'rand' function below for demonstration
11   // NOTE: When using a normal 'rand' function, the actual result may vary
12   let shuffled = nums.shuffle(rand=_ => 1)
13   inspect(shuffled, content="[1, 3, 4, 5, 2]")
14 }
```

Sorted Set

A mutable set backed by a red-black tree.

Usage

Create

You can create an empty SortedSet or a SortedSet from other containers.

```

1  ///|
2  test {
3    let _set1 : @sorted_set.SortedSet[Int] = @sorted_set.new()
4    let _set2 = @sorted_set.singleton(1)
5    let _set3 = @sorted_set.from_array([1])
6
7  }

```

Container Operations

Add an element to the SortedSet in place.

```

1  ///|
2  test {
3    let set4 = @sorted_set.from_array([1, 2, 3, 4])
4    set4.add(5) // ()
5    let set6 = @sorted_set.from_array([1, 2, 3, 4, 5])
6    assert_eq(set6.to_array(), [1, 2, 3, 4, 5])
7  }

```

Remove an element from the SortedSet in place.

```

1  ///|
2  test {
3    let set = @sorted_set.from_array([3, 8, 1])
4    set.remove(8) // ()
5    let set7 = @sorted_set.from_array([1, 3])
6    assert_eq(set7.to_array(), [1, 3])
7  }

```

Whether an element is in the set.

```

1  ///|
2  test {
3    let set = @sorted_set.from_array([1, 2, 3, 4])
4    assert_eq(set.contains(1), true)
5    assert_eq(set.contains(5), false)
6  }

```

Iterates over the elements in the set.

```

1  ///|
2  test {
3    let arr = []
4    @sorted_set.from_array([1, 2, 3, 4]).each(v => arr.push(v))
5    assert_eq(arr, [1, 2, 3, 4])
6  }

```

Get the size of the set.

```

1  ///|
2  test {
3    let set = @sorted_set.from_array([1, 2, 3, 4])
4    assert_eq(set.size(), 4)
5  }

```


Whether the set is empty.

```
1  ///|
2  test {
3    let set : @sorted_set.SortedSet[Int] = @sorted_set.new()
4    assert_eq(set.is_empty(), true)
5  }
```

Set Operations

Union, intersection and difference of two sets. They return a new set that does not overlap with the original sets in memory.

```
1  ///|
2  test {
3    let set1 = @sorted_set.from_array([3, 4, 5])
4    let set2 = @sorted_set.from_array([4, 5, 6])
5    let set3 = set1.union(set2)
6    assert_eq(set3.to_array(), [3, 4, 5, 6])
7    let set4 = set1.intersection(set2)
8    assert_eq(set4.to_array(), [4, 5])
9    let set5 = set1.difference(set2)
10   assert_eq(set5.to_array(), [3])
11 }
```

Determine the inclusion and separation relationship between two sets.

```
1  ///|
2  test {
3    let set1 = @sorted_set.from_array([1, 2, 3])
4    let set2 = @sorted_set.from_array([7, 2, 9, 4, 5, 6, 3, 8, 1])
5    assert_eq(set1.subset(set2), true)
6    let set3 = @sorted_set.from_array([4, 5, 6])
7    assert_eq(set1.disjoint(set3), true)
8  }
```

Stringify

SortedSet implements to_string (i.e. Show trait), which allows you to directly output it.

```
1  ///|
2  test {
3    let set = @sorted_set.from_array([1, 2, 3])
4    assert_eq(set.to_string(), "@sorted_set.from_array([1, 2, 3])")
5  }
```

int

The moonbitlang/core/int package provides essential operations on 32-bit integers.

Basic Operations

This section shows the basic operations available for integers:

```
1  ///|
2  test "basic int operations" {
3    // Get absolute value
4    inspect(@int.abs(-42), content="42")
5    inspect(@int.abs(42), content="42")
6
7    // Access min/max values
8    inspect(@int.min_value, content="-2147483648")
9    inspect(@int.max_value, content="2147483647")
10 }
```

Byte Conversion

The package provides methods to convert integers to their byte representation in both big-endian and little-endian formats:

```
1  ///|
2  test "byte conversions" {
3    let num = 258 // 0x0102 in hex
4
5    // Big-endian conversion (most significant byte first)
6    let be_bytes = num.to_be_bytes()
7    inspect(
8      be_bytes.to_string(),
9      content=(
10        #|b"\x00\x00\x01\x02"
11      ),
12    )
13
14    // Little-endian conversion (least significant byte first)
15    let le_bytes = num.to_le_bytes()
16    inspect(
17      le_bytes.to_string(),
18      content=(
19        #|b"\x02\x01\x00\x00"
20      ),
21    )
22 }
```

Method Syntax

All operations are also available using method syntax for better readability:

```

1  ///|
2  test "method syntax" {
3      let n = -42
4
5      // Using method syntax
6      inspect(n.abs(), content="42")
7
8      // Byte conversions using method syntax
9      let be = n.to_be_bytes()
10     let le = n.to_le_bytes()
11     inspect(
12         be.to_string(),
13         content=(
14             #|b"\xff\xff\xff\xd6"
15         ),
16     )
17     inspect(
18         le.to_string(),
19         content=(
20             #|b"\xd6\xff\xff\xff"
21         ),
22     )
23 }

```

The package provides the foundations for 32-bit integer operations in MoonBit, essential for any numeric computation.

int64

This package provides operations for working with 64-bit signed integers (Int64) in MoonBit.

Basic Operations

Int64 values can be created from regular 32-bit integers using `from_int`. The package also provides constants for the maximum and minimum values representable by Int64.

```

1  ///|
2  test "basic operations" {
3      let i : Int64 = -12345L // Int64 literal
4      // You can also convert from an 'Int' like so:
5      inspect(@int64.from_int(-12345) == i, content="true")
6
7      // Max and min values
8      inspect(@int64.max_value, content="9223372036854775807")
9      inspect(@int64.min_value, content="-9223372036854775808")
10
11     // Absolute value
12     inspect(@int64.abs(i), content="12345")
13 }

```

Binary Representation

The package provides functions to convert Int64 values to their binary representation in both big-endian and little-endian byte order:

```
1  ///|
2  test "binary conversion" {
3    let x = 258L // Int64 value of 258
4    let be_bytes = x.to_be_bytes()
5    let le_bytes = x.to_le_bytes()
6
7    // Convert to String for inspection
8    inspect(
9      be_bytes.to_string(),
10     content=(
11       #|b"\x00\x00\x00\x00\x00\x00\x01\x02"
12     ),
13   )
14   inspect(
15     le_bytes.to_string(),
16     content=(
17       #|b"\x02\x01\x00\x00\x00\x00\x00\x00"
18     ),
19   )
20
21   // We can verify they represent the same number but in different byte
22   let len = be_bytes.length()
23   inspect(len, content="8")
24 }
```

Method-Style Usage

All operations are also available as methods on Int64 values:

```
1  ///|
2  test "method style" {
3    let x = -42L
4
5    // Using method syntax for absolute value
6    inspect(x.abs(), content="42")
7
8    // Binary conversions as methods
9    inspect(
10     x.to_be_bytes(),
11     content=(
12       #|b"\xff\xff\xff\xff\xff\xff\x06"
13     ),
14   )
15 }
```

Note that Int64 implements the Hash trait, allowing it to be used as keys in hash maps and members of hash sets.

unit

The unit package provides functionality for working with the singleton type `Unit`, which represents computations that produce side effects but return no meaningful value. This is a fundamental type in functional programming for operations like I/O, logging, and state modifications.

Understanding Unit Type

The `Unit` type has exactly one value: `()`. This might seem trivial, but it serves important purposes in type systems:

- **Side Effect Indication:** Functions returning `Unit` signal they're called for side effects
- **Placeholder Type:** Used when a type parameter is needed but no meaningful value exists
- **Functional Programming:** Represents "no useful return value" without using `null` or exceptions
- **Interface Consistency:** Maintains uniform function signatures in generic contexts

Unit Value Creation

The unit value can be created in multiple ways:

```
1  ///|
2  test "unit construction" {
3    // Direct literal syntax
4    let u1 = ()
5
6    // Via default constructor
7    let u2 = @unit.default()
8    fn println(_ : String) {
9
10   }
11   // All unit values are identical
12   inspect(u1 == u2, content="true")
13
14   // Common pattern: functions that return unit
15   fn log_message(msg : String) -> Unit {
16     // In real code, this would write to a log
17     println(msg)
18     () // Explicit unit return
19   }
20
21   let result = log_message("Hello, world!")
22   inspect(result, content="()")
23 }
```

Working with Side-Effect Functions

Functions that return Unit are typically called for their side effects:

```
1  ///|
2  test "side effect patterns" {
3    let numbers = [1, 2, 3, 4, 5]
4    fn println(_ : Int) {
5
6    }
7    // Processing for side effects (printing, logging, etc.)
8    let processing_result = numbers.fold(init=(), fn(_acc, n) {
9      // Simulated side effect
10     if n % 2 == 0 {
11       // Would print or log in real code
12       println(n)
13     }
14     () // Return unit to continue fold
15   })
16   inspect(processing_result, content="()")
17
18   // Using each for side effects (more idiomatic)
19   numbers.each(fn(n) { if n % 2 == 0 { println(n) } })
20 }
```

String Representation and Debugging

Unit values have a standard string representation for debugging:

```
1  ///|
2  test "unit string conversion" {
3    let u = ()
4    inspect(u.to_string(), content="()")
5
6    // Useful for debugging function results
7    fn perform_operation() -> Unit {
8      // Some side effect operation
9      ()
10   }
11
12   let result = perform_operation()
13   let debug_msg = "Operation completed: \{result}"
14   inspect(debug_msg, content="Operation completed: ()")
15 }
```

Generic Programming with Unit

Unit is particularly useful in generic contexts where you need to represent "no meaningful value":

```

1  ///|
2  test "generic unit usage" {
3    // Simulating a function that processes data and optionally returns a
4    let items = [1, 2, 3, 4, 5]
5
6    // Process for side effects only
7    items.each(fn(x) {
8      // Side effect: processing each item
9      let processed = x * 2
10     assert_true(processed > 0) // Simulated processing validation
11   })
12
13   // Unit represents successful completion without meaningful return value
14   let completion_status = ()
15   inspect(completion_status, content="()")
16
17   // Unit is useful in Result types for operations that succeed but return a value
18   let operation_result : Result[Unit, String] = Ok(())
19   inspect(operation_result, content="Ok(())")
20 }

```

Built-in Trait Implementations

Unit implements essential traits for seamless integration with MoonBit's type system:

```

1  ///|
2  test "unit trait implementations" {
3    let u1 = ()
4    let u2 = ()
5
6    // Equality: all unit values are equal
7    inspect(u1 == u2, content="true")
8
9    // Comparison: all unit values compare as equal
10   inspect(u1.compare(u2), content="0")
11
12   // Hashing: consistent hash values
13   let h1 = u1.hash()
14   let h2 = u2.hash()
15   inspect(h1 == h2, content="true")
16
17   // Default instance
18   let u3 = Unit::default()
19   inspect(u3 == u1, content="true")
20 }

```

Practical Use Cases

Result Accumulation

```

1  ///|
2  test "result accumulation" {
3    // Accumulating side effects without meaningful return values
4    let operations = [
5      fn() { () }, // Operation 1
6      fn() { () }, // Operation 2
7      fn() { () },
8    ] // Operation 3
9    let final_result = operations.fold(init=(), fn(acc, operation) {
10      operation() // Execute the operation
11      acc // Accumulate unit values
12    })
13    inspect(final_result, content="()")
14  }

```

Builder Pattern Termination

```

1  ///|
2  test "builder pattern" {
3    // Simulating a builder pattern where build() returns Unit
4    let settings = ["debug=true", "timeout=30"]
5
6    // Build operation returns Unit after applying configuration
7    fn apply_config(config_list : Array[String]) -> Unit {
8      // In real code: apply configuration settings
9      let _has_settings = config_list.length() > 0
10     () // Unit indicates successful completion
11   }
12
13   let result = apply_config(settings)
14   inspect(result, content="()")
15 }

```

The Unit type provides essential functionality for representing "no meaningful return value" in a type-safe way, enabling clean functional programming patterns and consistent interfaces across MoonBit code.

BigInt Package Documentation

This package provides arbitrary-precision integer arithmetic through the BigInt type. BigInt allows you to work with integers of unlimited size, making it perfect for cryptographic operations, mathematical computations, and any scenario where standard integer types are insufficient.

Creating BigInt Values

There are several ways to create BigInt values:


```

1  ///|
2  test "creating bigint values" {
3    // From integer literals with 'N' suffix
4    let big1 = 12345678901234567890N
5    inspect(big1, content="12345678901234567890")
6
7    // From regular integers
8    let big2 = @bigint.BigInt::from_int(42)
9    inspect(big2, content="42")
10
11   // From Int64 values
12   let big3 = @bigint.BigInt::from_int64(9223372036854775807L)
13   inspect(big3, content="9223372036854775807")
14
15   // From strings
16   let big4 = @bigint.BigInt::from_string("1234567890123456789012345678901234567890")
17   inspect(big4, content="1234567890123456789012345678901234567890")
18
19   // From hexadecimal strings
20   let big5 = @bigint.BigInt::from_hex("1a2b3c4d5e6f")
21   inspect(big5, content="28772997619311")
22 }

```

Basic Arithmetic Operations

BigInt supports all standard arithmetic operations:

```

1  ///|
2  test "arithmetic operations" {
3    let a = 123456789012345678901234567890N
4    let b = 987654321098765432109876543210N
5
6    // Addition
7    let sum = a + b
8    inspect(sum, content="1111111110111111111011111111100")
9
10   // Subtraction
11   let diff = b - a
12   inspect(diff, content="864197532086419753208641975320")
13
14   // Multiplication
15   let product = @bigint.BigInt::from_int(123) * @bigint.BigInt::from_int(123)
16   inspect(product, content="56088")
17
18   // Division
19   let quotient = @bigint.BigInt::from_int(1000) / @bigint.BigInt::from_int(1000)
20   inspect(quotient, content="142")
21
22   // Modulo
23   let remainder = @bigint.BigInt::from_int(1000) % @bigint.BigInt::from_int(1000)
24   inspect(remainder, content="6")
25
26   // Negation
27   let neg = -a
28   inspect(neg, content="-123456789012345678901234567890")
29 }

```

Comparison Operations

Compare BigInt values with each other and with regular integers:

```

1  ///|
2  test "comparisons" {
3    let big = 12345N
4    let small = 123N
5
6    // BigInt to BigInt comparison
7    inspect(big > small, content="true")
8    inspect(big == small, content="false")
9    inspect(small < big, content="true")
10
11   // BigInt to Int comparison
12   inspect(big.equal_int(12345), content="true")
13   inspect(big.compare_int(12345), content="0")
14   inspect(big.compare_int(1000), content="1") // greater than
15   inspect(small.compare_int(200), content="-1") // less than
16
17   // BigInt to Int64 comparison
18   let big64 = @bigint.BigInt::from_int64(9223372036854775807L)
19   inspect(big64.equal_int64(9223372036854775807L), content="true")
20 }

```

Bitwise Operations

BigInt supports bitwise operations for bit manipulation:

```
1  ///|
2  test "bitwise operations" {
3    let a = 0b11110000N // 240 in decimal
4    let b = 0b10101010N // 170 in decimal
5
6    // Bitwise AND
7    let and_result = a & b
8    inspect(and_result, content="160") // 0b10100000
9
10   // Bitwise OR
11   let or_result = a | b
12   inspect(or_result, content="250") // 0b11111010
13
14   // Bitwise XOR
15   let xor_result = a ^ b
16   inspect(xor_result, content="90") // 0b01011010
17
18   // Bit length
19   let big_num = 255N
20   inspect(big_num.bit_length(), content="8")
21
22   // Count trailing zeros
23   let with_zeros = 1000N // Has trailing zeros in binary
24   let ctz = with_zeros.ctz()
25   inspect(ctz >= 0, content="true")
26 }
```

Power and Modular Arithmetic

BigInt provides efficient power and modular exponentiation:

```

1  ///|
2  test "power operations" {
3    // Basic power
4    let base = 2N
5    let exponent = 10N
6    let power = base.pow(exponent)
7    inspect(power, content="1024")
8
9    // Modular exponentiation (useful for cryptography)
10   let base2 = 3N
11   let exp2 = 5N
12   let modulus = 7N
13   let mod_power = base2.pow(exp2, modulus~)
14   inspect(mod_power, content="5") // (3^5) % 7 = 243 % 7 = 5
15
16   // Large modular exponentiation (optimized for speed)
17   let large_base = 123N
18   let large_exp = 20N
19   let large_mod = 1000007N
20   let result = large_base.pow(large_exp, modulus=large_mod)
21   inspect(result, content="378446") // (123^20) % 1000007
22 }

```

String and Hexadecimal Conversion

Convert BigInt to and from various string representations:

```

1  ///|
2  test "string conversions" {
3    let big = 255N
4
5    // Decimal string
6    let decimal = big.to_string()
7    inspect(decimal, content="255")
8
9    // Hexadecimal (lowercase)
10   let hex_lower = big.to_hex()
11   inspect(hex_lower, content="FF")
12
13   // Hexadecimal (uppercase)
14   let hex_upper = big.to_hex(uppercase=true)
15   inspect(hex_upper, content="FF")
16
17   // Parse from hex
18   let from_hex = @bigint.BigInt::from_hex("deadbeef")
19   inspect(from_hex, content="3735928559")
20
21   // Round-trip conversion
22   let original = 98765432109876543210N
23   let as_string = original.to_string()
24   let parsed_back = @bigint.BigInt::from_string(as_string)
25   inspect(original == parsed_back, content="true")
26 }

```

Byte Array Conversion

Convert BigInt to and from byte arrays:

```
1  ///|
2  test "byte conversions" {
3    let big = 0x123456789abcdefN
4
5    // Convert to bytes
6    let bytes = big.to_octets()
7    inspect(bytes.length() > 0, content="true")
8
9    // Convert from bytes (positive number)
10   let from_bytes = @bigint.BigInt::from_octets(bytes)
11   inspect(from_bytes == big, content="true")
12
13   // Convert with specific length
14   let fixed_length = @bigint.BigInt::from_int(255).to_octets(length=4)
15   inspect(fixed_length.length(), content="4")
16
17   // Negative numbers
18   // let negative = -big
19   // let neg_bytes = negative.to_octets()
20   // to_octets does not accept negative numbers
21   // let neg_from_bytes = @bigint.BigInt::from_octets(neg_bytes, signum=)
22   // inspect(neg_from_bytes == negative, content="true")
23 }
```

Type Conversions

Convert BigInt to standard integer types:


```

1  ///|
2  test "utility functions" {
3      let zero = 0N
4      let positive = 42N
5      let negative = -42N
6
7      // Check if zero
8      inspect(zero.is_zero(), content="true")
9      inspect(positive.is_zero(), content="false")
10
11     // Sign testing through comparison
12     inspect(positive > zero, content="true")
13     inspect(negative < zero, content="true")
14     inspect(zero == zero, content="true")
15 }

```

Use Cases and Applications

BigInt is particularly useful for:

- **Cryptography:** RSA encryption, digital signatures, and key generation
- **Mathematical computations:** Factorial calculations, Fibonacci sequences, prime number testing
- **Financial calculations:** High-precision monetary computations
- **Scientific computing:** Large integer calculations in physics and chemistry
- **Data processing:** Handling large numeric IDs and checksums

Performance Considerations

- BigInt operations are slower than regular integer operations due to arbitrary precision
- Addition and subtraction are generally fast
- Multiplication and division become slower with larger numbers
- Modular exponentiation is optimized for cryptographic use cases
- String conversions can be expensive for very large numbers

Best Practices

- **Use regular integers when possible:** Only use BigInt when you need arbitrary precision
- **Cache string representations:** If you need to display the same BigInt multiple times
- **Use modular arithmetic:** For cryptographic applications, always use modular exponentiation
- **Be careful with conversions:** Converting very large BigInt to regular integers will truncate
- **Consider memory usage:** Very large BigInt values consume more memory

Bench Package Documentation

This package provides benchmarking utilities for measuring the performance of MonBit code. It includes functions for timing code execution, collecting statistics, and generating performance reports.

Basic Benchmarking

Use the `single_bench` function to benchmark individual operations:

```
1  ///|
2  #skip("slow tests")
3  test "basic benchmarking" {
4      fn simple_calc(n : Int) -> Int {
5          n * 2 + 1
6      }
7      // Benchmark a simple computation
8      let summary = @bench.single_bench(name="simple_calc", fn() {
9          ignore(simple_calc(5))
10     })
11
12     // The benchmark ran successfully (we can't inspect exact timing)
13     inspect(summary.to_json().stringify().length() > 0, content="true")
14 }
```

Benchmark Collection

Use the `T` type to collect multiple benchmarks:


```

1  ///|
2  #skip("slow tests")
3  test "benchmark collection" {
4    let bencher = @bench.new()
5
6    // Add multiple benchmarks to the collection
7    bencher.bench(name="array_creation", fn() {
8      let arr = Array::new()
9      for i in 0..<5 {
10        arr.push(i)
11      }
12    })
13    bencher.bench(name="array_iteration", fn() {
14      let arr = [1, 2, 3, 4, 5]
15      let mut sum = 0
16      for x in arr {
17        sum = sum + x
18      }
19    })
20
21    // Generate benchmark report
22    let report = bencher.dump_summaries()
23    inspect(report.length() > 0, content="true")
24  }

```

Benchmarking Different Algorithms

Compare the performance of different implementations:

```

1  ///|
2  #skip("slow tests")
3  test "algorithm comparison" {
4      let bencher = @bench.new()
5
6      // Benchmark linear search
7      bencher.bench(name="linear_search", fn() {
8          let arr = [1, 2, 3, 4, 5]
9          let target = 3
10         let mut found = false
11         for x in arr {
12             if x == target {
13                 found = true
14                 break
15             }
16         }
17         ignore(found)
18     })
19
20     // Benchmark using built-in contains (likely optimized)
21     bencher.bench(name="builtin_contains", fn() {
22         let arr = [1, 2, 3, 4, 5]
23         ignore(arr.contains(3))
24     })
25     let results = bencher.dump_summaries()
26     inspect(results.length() > 10, content="true") // Should have benchmarked
27 }

```

Data Structure Benchmarks

Benchmark different data structure operations:

```

1  ///|
2  #skip("slow tests")
3  test "data structure benchmarks" {
4    let bencher = @bench.new()
5
6    // Benchmark Array operations
7    bencher.bench(name="array_append", fn() {
8      let arr = Array::new()
9      for i in 0..<5 {
10        arr.push(i)
11      }
12    })
13
14    // Benchmark FixedArray access
15    bencher.bench(name="fixedarray_access", fn() {
16      let arr = [0, 1, 2, 3, 4]
17      let mut sum = 0
18      for i in 0..<arr.length() {
19        sum = sum + arr[i]
20      }
21      ignore(sum)
22    })
23    let report = bencher.dump_summaries()
24    inspect(report.length() > 50, content="true") // Should have benchmark
25  }

```

String Operations Benchmarking

Measure string manipulation performance:

```

1  ///|
2  #skip("slow tests")
3  test "string benchmarks" {
4    let bencher = @bench.new()
5
6    // Benchmark string concatenation
7    bencher.bench(name="string_concat", fn() {
8      let mut result = ""
9      for i in 0..<5 {
10        result = result + "x"
11      }
12    })
13
14    // Benchmark StringBuilder (should be faster)
15    bencher.bench(name="stringbuilder", fn() {
16      let builder = StringBuilder::new()
17      for i in 0..<5 {
18        builder.write_string("x")
19      }
20      ignore(builder.to_string())
21    })
22    let results = bencher.dump_summaries()
23    inspect(results.length() > 50, content="true") // Should have benchmark
24  }

```

Memory Usage Prevention

Use keep to prevent compiler optimizations from eliminating benchmarked code:

```
1  ///|
2  #skip("slow tests")
3  test "preventing optimization" {
4    let bencher = @bench.new()
5    bencher.bench(name="with_keep", fn() {
6      let result = Array::makei(5, fn(i) { i * i })
7      // Prevent the compiler from optimizing away the computation
8      bencher.keep(result)
9    })
10   let report = bencher.dump_summaries()
11   inspect(report.length() > 30, content="true") // Should have benchmarked code
12 }
```

Iteration Count Control

Control the number of benchmark iterations:

```
1  ///|
2  #skip("slow tests")
3  test "iteration control" {
4    let bencher = @bench.new()
5
6    // Run with more iterations for more stable results
7    bencher.bench(
8      name="stable_benchmark",
9      fn() {
10        let arr = [1, 2, 3, 4, 5]
11        let sum = arr.fold(init=0, fn(acc, x) { acc + x })
12        ignore(sum)
13      },
14      count=20,
15    )
16
17    // Run with fewer iterations for quick testing
18    bencher.bench(
19      name="quick_benchmark",
20      fn() {
21        let mut result = 0
22        for i in 0..<10 {
23          result = result + i
24        }
25        ignore(result)
26      },
27      count=2,
28    )
29    let results = bencher.dump_summaries()
30    inspect(results.length() > 50, content="true") // Should have benchmarked code
31 }
```

Benchmarking Best Practices

1. Isolate What You're Measuring

```
1  ///|
2  #skip("slow tests")
3  test "isolation example" {
4    let bencher = @bench.new()
5
6    // Good: Measure only the operation of interest
7    let data = Array::makei(10, fn(i) { i }) // Setup outside benchmark
8    bencher.bench(name="array_sum", fn() {
9      let mut sum = 0
10     for x in data {
11       sum = sum + x
12     }
13     bencher.keep(sum) // Prevent optimization
14   })
15   let results = bencher.dump_summaries()
16   inspect(results.length() > 0, content="true")
17 }
```

2. Warm Up Before Measuring

```
1  ///|
2  #skip("slow tests")
3  test "warmup example" {
4    let bencher = @bench.new()
5    fn expensive_operation() -> Int {
6      let mut result = 0
7      for i in 0..<5 {
8        result = result + i * i
9      }
10     result
11   }
12
13   // Warm up the function (not measured)
14   for _ in 0..<5 {
15     ignore(expensive_operation())
16   }
17
18   // Now benchmark the warmed-up function
19   bencher.bench(name="warmed_up", fn() {
20     let result = expensive_operation()
21     bencher.keep(result)
22   })
23   let report = bencher.dump_summaries()
24   inspect(report.length() > 30, content="true") // Should have benchmarked
25 }
```

3. Use Meaningful Names

```

1  ///|
2  #skip("slow tests")
3  test "meaningful names" {
4    let bencher = @bench.new()
5
6    // Good: Descriptive names that explain what's being measured
7    bencher.bench(name="array_insert_10_items", fn() {
8      let arr = Array::new()
9      for i in 0..<10 {
10       arr.push(i * 2)
11     }
12     bencher.keep(arr)
13   })
14   bencher.bench(name="array_search_sorted_10", fn() {
15     let arr = Array::makei(10, fn(i) { i })
16     let result = arr.contains(5) // Linear search in this case
17     bencher.keep(result)
18   })
19   let results = bencher.dump_summaries()
20   inspect(results.length() > 50, content="true") // Should have benchmark
21 }

```

Performance Analysis

The benchmark results include statistical information:

- **Timing measurements:** Microsecond precision timing
- **Statistical analysis:** Median, percentiles, and outlier detection
- **Batch sizing:** Automatic adjustment for stable measurements
- **JSON output:** Machine-readable results for analysis

Integration with Testing

Benchmarks can be integrated into your testing workflow:

```

1  ///|
2  #skip("slow tests")
3  test "performance regression test" {
4      let bencher = @bench.new()
5
6      // Benchmark a critical path
7      bencher.bench(name="critical_algorithm", fn() {
8          let data = [5, 2, 8, 1, 9, 3, 7, 4, 6]
9          let sorted = Array::new()
10         for x in data {
11             sorted.push(x)
12         }
13         sorted.sort()
14         bencher.keep(sorted)
15     })
16     let results = bencher.dump_summaries()
17     // In a real scenario, you might parse results and assert performance
18     inspect(results.length() > 50, content="true") // Should have substantial
19 }

```

Common Benchmarking Patterns

- **Before/After comparisons:** Benchmark code before and after optimizations
- **Algorithm comparison:** Compare different implementations of the same functionality
- **Scaling analysis:** Benchmark with different input sizes
- **Memory vs. speed tradeoffs:** Compare memory-efficient vs. speed-optimized approaches
- **Platform differences:** Compare performance across different targets (JS, WASM, native)

Tips for Accurate Benchmarks

- Run benchmarks multiple times and look for consistency
- Be aware of system load and other processes affecting timing
- Use appropriate iteration counts (more for stable results, fewer for quick feedback)
- Measure what matters to your use case
- Consider both average case and worst case performance
- Profile memory usage separately if memory performance is important

The bench package provides essential tools for performance analysis and optimization in MoonBit applications.

cmp

This package provides utility functions for comparing values.

Generic Comparison Functions

The library provides generic comparison functions that work with any type implementing the Compare trait:

```
1  ///|
2  test "generic comparison" {
3    // Works with numbers
4    inspect(@cmp.maximum(3, 4), content="4")
5    inspect(@cmp.minimum(3, 4), content="3")
6  }
```

Comparison by Key

With `@cmp.maximum_by_key()` and `@cmp.minimum_by_key()`, it is possible to compare values based on arbitrary keys derived from them. This is particularly useful when you need to compare complex objects based on some specific aspect or field.

```
1  ///|
2  test "cmp_by_key" {
3    struct Person {
4      name : String
5      age  : Int
6    } derive(Show)
7
8    // Compare strings by their length
9    let s1 = "hello"
10   let s2 = "hi"
11   let longer = @cmp.maximum_by_key(s1, s2, String::length)
12   inspect(longer, content="hello")
13
14   // Compare structs by a specific field
15   let alice = { name: "Alice", age: 25 }
16   let bob   = { name: "Bob",   age: 30 }
17   let younger = @cmp.minimum_by_key(alice, bob, p => p.age)
18   inspect(younger, content="{name: \"Alice\", age: 25}")
19
20   // When keys are equal, the first argument is considered the minimum
21   let p1 = ("first", 1)
22   let p2 = ("second", 1)
23   let snd = (p : (_, _)) => p.1
24   assert_eq(@cmp.minimum_by_key(p1, p2, snd), p1)
25   assert_eq(@cmp.maximum_by_key(p1, p2, snd), p2)
26 }
```

math

This library provides common mathematical functions for floating-point arithmetic, trigonometry, and general numeric comparisons.

Constants

MoonBit math library provides the mathematical constant π


```

1  ///|
2  test "mathematical constants" {
3      inspect(@math.PI, content="3.141592653589793")
4  }

```

Basic Arithmetic Functions

Rounding Functions

Several functions are available for rounding numbers in different ways:

```

1  ///|
2  test "rounding functions" {
3      // Round to nearest integer
4      inspect(@math.round(3.7), content="4")
5      inspect(@math.round(-3.7), content="-4")
6
7      // Ceiling (round up)
8      inspect(@math.ceil(3.2), content="4")
9      inspect(@math.ceil(-3.2), content="-3")
10
11     // Floor (round down)
12     inspect(@math.floor(3.7), content="3")
13     inspect(@math.floor(-3.7), content="-4")
14
15     // Truncate (round toward zero)
16     inspect(@math.trunc(3.7), content="3")
17     inspect(@math.trunc(-3.7), content="-3")
18 }

```

Exponential and Logarithmic Functions

The library provides standard exponential and logarithmic operations:

```

1  ///|
2  test "exponential and logarithmic" {
3      // Exponential functions
4      inspect(@math.exp(1.0), content="2.718281828459045")
5      inspect(@math.expml(1.0), content="1.718281828459045")
6
7      // Natural logarithm
8      inspect(@math.ln(2.718281828459045), content="1")
9      inspect(@math.ln_lp(1.718281828459045), content="1")
10
11     // Other logarithm bases
12     inspect(@math.log2(8.0), content="3")
13     inspect(@math.log10(100.0), content="2")
14 }

```

Trigonometric Functions

Basic Trigonometric Functions

Standard trigonometric functions operating in radians:

```
1  ///|
2  test "basic trigonometry" {
3    // Basic trig functions
4    inspect(@math.sin(@math.PI / 2.0), content="1")
5    inspect(@math.cos(0.0), content="1")
6    inspect(@math.tan(@math.PI / 4.0), content="0.9999999999999999")
7
8    // Inverse trig functions
9    inspect(@math.asin(1.0), content="1.5707963267948966")
10   inspect(@math.acos(1.0), content="0")
11   inspect(@math.atan(1.0), content="0.7853981633974483")
12 }
```

Hyperbolic Functions

The library also includes hyperbolic functions and their inverses:

```
1  ///|
2  test "hyperbolic functions" {
3    // Hyperbolic functions
4    inspect(@math.sinh(1.0), content="1.1752011936438014")
5    inspect(@math.cosh(1.0), content="1.5430806348152437")
6    inspect(@math.tanh(1.0), content="0.7615941559557649")
7
8    // Inverse hyperbolic functions
9    inspect(@math.asinh(1.0), content="0.881373587019543")
10   inspect(@math.acosh(2.0), content="1.3169578969248166")
11   inspect(@math.atanh(0.5), content="0.5493061443340548")
12 }
```

Special Functions

Two-argument Functions

Some special mathematical functions taking two arguments:

```
1  ///|
2  test "special functions" {
3    // atan2 gives the angle in radians between the positive x-axis and the
4    inspect(@math.atan2(1.0, 1.0), content="0.7853981633974483")
5
6    // hypot computes sqrt(x*x + y*y) without intermediate overflow
7    inspect(@math.hypot(3.0, 4.0), content="5")
8
9    // Cube root
10   inspect(@math.cbrt(8.0), content="2")
11 }
```

ref

This package provides functionality for working with mutable references, allowing you to create sharable mutable values that can be modified safely.

Creating and Accessing References

References can be created using `@ref.new()`. The reference value can be accessed through the `val` field:

```
1  ///|
2  test "creating and accessing refs" {
3    let r1 = @ref.new(42)
4    inspect(r1.val, content="42")
5  }
```

Updating Reference Values

The update function allows modifying the contained value using a transformation function:

```
1  ///|
2  test "updating refs" {
3    let counter = @ref.new(0)
4    counter.update(x => x + 1)
5    inspect(counter.val, content="1")
6    counter.update(x => x * 2)
7    inspect(counter.val, content="2")
8  }
```

Mapping References

The map function transforms a reference while preserving the reference wrapper:

```
1  ///|
2  test "mapping refs" {
3    let num = @ref.new(10)
4    let doubled = num.map(x => x * 2)
5    inspect(doubled.val, content="20")
6    let squared = num.map(x => x * x)
7    inspect(squared.val, content="100")
8  }
```

Swapping Reference Values

You can exchange the values of two references using the swap function:

```

1  ///|
2  test "swapping refs" {
3    let r1 = @ref.new("first")
4    let r2 = @ref.new("second")
5    @ref.swap(r1, r2)
6    inspect(r1.val, content="second")
7    inspect(r2.val, content="first")
8  }

```

Temporary Value Protection

The `protect` function temporarily sets a reference to a value and restores it after executing a block:

```

1  ///|
2  test "protected updates" {
3    let state = @ref.new(100)
4    let mut middle = 0
5    let result = state.protect(50, () => {
6      middle = state.val
7      42
8    })
9    inspect(middle, content="50")
10   inspect(result, content="42")
11   inspect(state.val, content="100")
12 }

```

This is useful for temporarily modifying state that needs to be restored afterwards.

bytes

This package provides utilities for working with sequences of bytes, offering both mutable (`Bytes`) and immutable (`View`) representations.

Creating Bytes

You can create `Bytes` from various sources including arrays, fixed arrays, and iterators:

```

1  ///|
2  test "bytes creation" {
3    // Create from array of bytes
4    let arr = [b'h', b'e', b'l', b'l', b'o']
5    let bytes1 = @bytes.from_array(arr)
6    inspect(
7      bytes1,
8      content=(
9        #|b"\x68\x65\x6c\x6c\x6f"
10     ),
11   )
12
13   // Create from fixed array
14   let fixed = FixedArray::make(3, b'a')
15   let bytes2 = @bytes.of(fixed)
16   inspect(
17     bytes2,
18     content=(
19       #|b"\x61\x61\x61"
20     ),
21   )
22
23   // Create empty bytes
24   let empty = @bytes.default()
25   inspect(
26     empty,
27     content=(
28       #|b""
29     ),
30   )
31
32   // Create from iterator
33   let iter_bytes = @bytes.from_iter(arr.iter())
34   inspect(
35     iter_bytes,
36     content=(
37       #|b"\x68\x65\x6c\x6c\x6f"
38     ),
39   )
40 }

```

Converting Between Formats

Bytes can be converted to and from different formats:

```

1  ///|
2  test "bytes conversion" {
3      let original = [b'x', b'y', b'z']
4      let bytes = @bytes.from_array(original)
5
6      // Convert to array
7      let array = bytes.to_array()
8      inspect(array, content="[b'\\x78', b'\\x79', b'\\x7A']")
9
10     // Convert to fixed array
11     let fixed = bytes.to_fixedarray()
12     inspect(fixed, content="[b'\\x78', b'\\x79', b'\\x7A']")
13
14     // Convert to iterator and collect back
15     let collected = bytes.iter().to_array()
16     inspect(collected, content="[b'\\x78', b'\\x79', b'\\x7A']")
17 }

```

Working with Views

Views provide a way to work with portions of bytes and interpret them as various numeric types:

```

1  ///|
2  test "bytes view operations" {
3      // Create bytes with numeric data
4      let num_bytes = @bytes.from_array([0x12, 0x34, 0x56, 0x78])
5
6      // Create a view
7      let view = num_bytes[:]
8
9      // Get individual bytes
10     inspect(view[0], content="b'\\x12'")
11
12     // Interpret as integers (big-endian)
13     inspect(view.to_int_be(), content="305419896")
14
15     // Interpret as integers (little-endian)
16     inspect(view.to_int_le(), content="2018915346")
17
18     // Create a sub-view
19     let sub_view = view[1:3]
20     inspect(sub_view.length(), content="2")
21 }

```

Binary Data Interpretation

Views provide methods to interpret byte sequences as various numeric types in both little-endian and big-endian formats:

```

1  ///|
2  test "numeric interpretation" {
3    // Create test data
4    let int64_bytes = @bytes.from_array([
5      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x42,
6    ])
7    let int64_view = int64_bytes[:]
8    inspect(int64_view.to_int64_be(), content="66")
9    inspect(int64_view.to_uint64_le(), content="4755801206503243776")
10 }

```

Concatenation and Comparison

Bytes can be concatenated and compared:

```

1  ///|
2  test "bytes operations" {
3    let b1 = @bytes.from_array([b'a', b'b'])
4    let b2 = @bytes.from_array([b'c', b'd'])
5
6    // Concatenation
7    let combined = b1 + b2
8    inspect(
9      combined,
10     content=(
11       #|b"\x61\x62\x63\x64"
12     ),
13   )
14
15   // Comparison
16   let same = @bytes.from_array([b'a', b'b'])
17   let different = @bytes.from_array([b'x', b'y'])
18   inspect(b1 == same, content="true")
19   inspect(b1 == different, content="false")
20   inspect(b1 < b2, content="true")
21 }

```

char

This package provides a set of utilities for working with characters, focusing on character classification and validation.

Basic ASCII Classification

Functions for determining if a character belongs to various ASCII categories.

```

1  ///|
2  test "ascii classification" {
3      // Basic ASCII checks
4      inspect('A'.is_ascii(), content="true")
5      inspect('?.is_ascii(), content="false")
6
7      // Letter classification
8      inspect('Z'.is_ascii_alphabetic(), content="true")
9      inspect('1'.is_ascii_alphabetic(), content="false")
10
11     // Case classification
12     inspect('A'.is_ascii_uppercase(), content="true")
13     inspect('a'.is_ascii_uppercase(), content="false")
14     inspect('a'.is_ascii_lowercase(), content="true")
15     inspect('A'.is_ascii_lowercase(), content="false")
16 }

```

Number Classification

Functions for identifying digits in different number bases.

```

1  ///|
2  test "number classification" {
3      // Decimal digits
4      inspect('5'.is_ascii_digit(), content="true")
5      inspect('x'.is_ascii_digit(), content="false")
6
7      // Hexadecimal digits
8      inspect('F'.is_ascii_hexdigit(), content="true")
9      inspect('G'.is_ascii_hexdigit(), content="false")
10
11     // Octal digits
12     inspect('7'.is_ascii_octdigit(), content="true")
13     inspect('8'.is_ascii_octdigit(), content="false")
14
15     // Custom base digits
16     inspect('5'.is_digit(6U), content="true")
17     inspect('6'.is_digit(6U), content="false")
18
19     // General numeric characters
20     inspect('1'.is_numeric(), content="true")
21     inspect('A'.is_numeric(), content="false")
22 }

```

Special Characters

Functions for identifying whitespace, control characters and other special characters.


```

1  ///|
2  test "special characters" {
3    // Whitespace characters
4    inspect(' '.is_ascii_whitespace(), content="true")
5    inspect('\n'.is_whitespace(), content="true")
6
7    // Control characters
8    inspect('\u0000'.is_ascii_control(), content="true")
9    inspect('\u007F'.is_control(), content="true")
10
11   // Graphic and punctuation characters
12   inspect('!'.is_ascii_graphic(), content="true")
13   inspect(' '.is_ascii_graphic(), content="false")
14   inspect(',',.is_ascii_punctuation(), content="true")
15 }

```

Method Style Usage

All character classification functions can also be called as methods directly on characters.

```

1  ///|
2  test "method style" {
3    // Letter methods
4    let c = 'A'
5    inspect(c.is_ascii(), content="true")
6    inspect(c.is_ascii_alphabetic(), content="true")
7    inspect(c.is_ascii_uppercase(), content="true")
8
9    // Digit methods
10   let d = '7'
11   inspect(d.is_ascii_digit(), content="true")
12   inspect(d.is_digit(8U), content="true")
13   inspect(d.is_ascii_hexdigit(), content="true")
14
15   // Special character methods
16   let s = ' '
17   inspect(s.is_ascii_whitespace(), content="true")
18   inspect(s.is_whitespace(), content="true")
19 }

```

Deque

Deque is a double-ended queue implemented as a round-robin queue, supporting O(1) head or tail insertion and querying, just like double-ended queues in other languages(C++ `std::deque` / Rust `VecDeque`), here deque also supports random access.

Usage

Create

You can create a deque manually via the `new()` or construct it using the `of()`

.

```
1  ///|
2  test {
3      let _dv : @deque.Deque[Int] = @deque.new()
4      let _dv = @deque.of([1, 2, 3, 4, 5])
5
6  }
```

If you want to set the length at creation time to minimize expansion consumption, you can add parameter capacity to the `new()` function.

```
1  ///|
2  test {
3      let _dv : @deque.Deque[Int] = @deque.new(capacity=10)
4
5  }
```

Length & Capacity

A deque is an indefinite-length, auto-expandable datatype. You can use `length()` to get the number of elements and `capacity()` to get the current capacity.

```
1  ///|
2  test {
3      let dv = @deque.of([1, 2, 3, 4, 5])
4      assert_eq(dv.length(), 5)
5      assert_eq(dv.capacity(), 5)
6  }
```

Similarly, you can use the `is_empty` to determine whether the queue is empty.

```
1  ///|
2  test {
3      let dv : @deque.Deque[Int] = @deque.new()
4      assert_eq(dv.is_empty(), true)
5  }
```

You can use `reserve_capacity` to reserve capacity, ensures that it can hold at least the number of elements specified by the capacity argument.

```
1  ///|
2  test {
3      let dv = @deque.of([1])
4      dv.reserve_capacity(10)
5      assert_eq(dv.capacity(), 10)
6  }
```

Also, you can use `shrink_to_fit` to shrink the capacity of the deque.

```

1  ///|
2  test {
3      let dv = @deque.new(capacity=10)
4      dv.push_back(1)
5      dv.push_back(2)
6      dv.push_back(3)
7      assert_eq(dv.capacity(), 10)
8      dv.shrink_to_fit()
9      assert_eq(dv.capacity(), 3)
10 }

```

Front & Back & Get

You can use `front()` and `back()` to get the head and tail elements of the queue, respectively. Since the queue may be empty, their return values are both `Option`, or `None` if the queue is empty.

```

1  ///|
2  test {
3      let dv = @deque.of([1, 2, 3, 4, 5])
4      assert_eq(dv.front(), Some(1))
5      assert_eq(dv.back(), Some(5))
6  }

```

You can also use `get` to access elements of the queue directly, but be careful not to cross the boundaries!

```

1  ///|
2  test {
3      let dv = @deque.of([1, 2, 3, 4, 5])
4      assert_eq(dv[0], 1)
5      assert_eq(dv[4], 5)
6  }

```

Push & Set

Since the queue is bi-directional, you can use `push_front()` and `push_back()` to add values to the head or tail of the queue, respectively.

```

1  ///|
2  test {
3      let dv = @deque.of([1, 2, 3, 4, 5])
4      dv.push_front(6)
5      dv.push_front(7)
6      dv.push_back(8)
7      dv.push_back(9)
8      //now: 6 7 1 2 3 4 5 8 9
9  }

```

You can also use `Deque::set` or operator `[_]=` to set elements of the queue directly, but be careful not to cross the boundaries!

```

1  ///|
2  test {
3      let dv = @deque.of([1, 2, 3, 4, 5])
4      dv[0] = 5
5      assert_eq(dv[0], 5)
6  }

```

Pop

You can use `pop_front()` and `pop_back()` to pop the element at the head or tail of the queue, respectively, and like `[Front & Back](#Front & Back & Get)`, their return values are `Option`, loaded with the value of the element being popped.

```

1  ///|
2  test {
3      let dv = @deque.of([1, 2, 3, 4, 5])
4      let _back = dv.pop_back() // Some(5)
5      assert_eq(dv.back(), Some(4))
6      let _front = dv.pop_front() //Some(1)
7      assert_eq(dv.front(), Some(2))
8      assert_eq(dv.length(), 3)
9  }

```

If you only want to pop an element without getting the return value, you can use `unsafe_pop_front()` with `unsafe_pop_back()`. These two functions will panic if the queue is empty.

```

1  ///|
2  test {
3      let dv = @deque.of([1, 2, 3, 4, 5])
4      dv.unsafe_pop_front()
5      assert_eq(dv.front(), Some(2))
6      dv.unsafe_pop_back()
7      assert_eq(dv.back(), Some(4))
8  }

```

Clear

You can use `clear` to clear a deque. But note that the memory it already occupies does not change.

```

1  ///|
2  test {
3      let dv = @deque.of([1, 2, 3, 4, 5])
4      dv.clear()
5      assert_eq(dv.is_empty(), true)
6  }

```

Equal

deque supports comparing them directly using `equal`.

```

1  ///|
2  test {
3      let dqa = @deque.of([1, 2, 3, 4, 5])
4      let dqb = @deque.of([1, 2, 3, 4, 5])
5      assert_eq(dqa, dqb)
6  }

```

Iter & Map

deque supports vector-like iter/iteri/map/mapi functions and their inverse forms

```

1  ///|
2  test {
3      let dv = @deque.of([1, 2, 3, 4, 5])
4      let arr = []
5      dv.each(elem => arr.push(elem))
6      assert_eq(arr, [1, 2, 3, 4, 5])
7      let arr2 = []
8      dv.eachi((i, _elem) => arr2.push(i))
9      assert_eq(arr2, [0, 1, 2, 3, 4])
10     let arr3 = []
11     let _ = dv.map(elem => arr3.push(elem + 1))
12     assert_eq(arr3, [2, 3, 4, 5, 6])
13     let arr4 = []
14     let _ = dv.mapi((i, elem) => arr4.push(elem + i))
15     assert_eq(arr4, [1, 3, 5, 7, 9])
16 }

```

Search & Contains

You can use `contains()` to find out if a value is in the deque, or `search()` to find its index in the deque.

```

1  ///|
2  test {
3      let dv = @deque.of([1, 2, 3, 4, 5])
4      assert_eq(dv.contains(1), true)
5      assert_eq(dv.contains(6), false)
6      assert_eq(dv.search(1), Some(0))
7      assert_eq(dv.search(6), None)
8  }

```

Error Package Documentation

This package provides utilities for working with MoonBit's error handling system, including implementations of `Show` and `ToJson` traits for the built-in `Error` type.

Basic Error Usage

MoonBit uses a structured error system with `raise` and `try` constructs:

```

1  ///|
2  test "basic error handling" {
3      fn divide(a : Int, b : Int) -> Int raise {
4          if b == 0 {
5              raise Failure("Division by zero")
6          } else {
7              a / b
8          }
9      }
10
11     // Successful operation
12     let result1 = try! divide(10, 2)
13     inspect(result1, content="5")
14
15     // Handle error with try?
16     let result2 = try? divide(10, 0)
17     inspect(result2, content="Err(Failure(\"Division by zero\"))")
18 }

```

Custom Error Types

Define custom error types using suberror:

```

1  ///|
2  suberror ValidationError String
3
4  ///|
5  suberror NetworkError String
6
7  ///|
8  test "custom errors" {
9      fn validate_email(email : String) -> String raise ValidationError {
10         if email.length() > 5 { // Simple validation without string methods
11             email
12         } else {
13             raise ValidationError("Invalid email format")
14         }
15     }
16
17     fn fetch_data(url : String) -> String raise NetworkError {
18         if url.length() > 10 { // Simple validation
19             "data"
20         } else {
21             raise NetworkError("Invalid URL")
22         }
23     }
24
25     // Test validation error
26     let email_result = try? validate_email("short")
27     match email_result {
28         Ok(_) => inspect(false, content="true")
29         Err(_) => inspect(true, content="true")
30     }
31
32     // Test network error
33     let data_result = try? fetch_data("short")
34     match data_result {
35         Ok(_) => inspect(false, content="true")
36         Err(_) => inspect(true, content="true")
37     }
38 }

```

Error Display and JSON Conversion

The error package provides Show and ToJson implementations:

```

1  ///|
2  suberror MyError Int derive(ToJson)
3
4  ///|
5  test "error display and json" {
6    let error : Error = MyError(42)
7
8    // Error can be displayed as string
9    let error_string = error.to_string()
10   inspect(error_string.length() > 0, content="true")
11
12   // Error can be converted to JSON
13   let error_json = error.to_json()
14   inspect(error_json, content="Array([String(\"MyError\"), Number(42)])")
15 }

```

Error Propagation and Handling

Handle errors at different levels of your application:

```

1  ///|
2  suberror ParseError String
3
4  ///|
5  suberror FileError String
6
7  ///|
8  test "error propagation" {
9    fn parse_number(s : String) -> Int raise ParseError {
10      if s == "42" {
11        42
12      } else {
13        raise ParseError("Invalid number: " + s)
14      }
15    }
16
17    fn read_and_parse(content : String) -> Int raise {
18      parse_number(content) catch {
19        ParseError(msg) => raise FileError("Parse failed: " + msg)
20      }
21    }
22
23    // Success case
24    let result1 = try! read_and_parse("42")
25    inspect(result1, content="42")
26
27    // Error propagation
28    let result2 = try? read_and_parse("invalid")
29    match result2 {
30      Ok(_) => inspect(false, content="true")
31      Err(_) => inspect(true, content="true")
32    }
33 }

```


Resource Management with Finally

Use protect functions for resource cleanup:

```
1  ///|
2  suberror ResourceError String
3
4  ///|
5  test "resource management" {
6    fn risky_operation() -> String raise ResourceError {
7      raise ResourceError("Something went wrong")
8    }
9
10   // Simple resource management pattern
11   fn use_resource() -> String raise {
12     // Acquire resource (simulated)
13     risky_operation() catch {
14       ResourceError(_) =>
15         // Cleanup happens here
16         raise Failure("Operation failed after cleanup")
17     }
18   }
19
20   let result = try? use_resource()
21   match result {
22     Ok(_) => inspect(false, content="true")
23     Err(_) => inspect(true, content="true")
24   }
25 }
```

Error Composition

Combine multiple error-producing operations:

```

1  ///|
2  suberror ConfigError String
3
4  ///|
5  suberror DatabaseError String
6
7  ///|
8  test "error composition" {
9      fn load_config() -> String raise ConfigError {
10         if true {
11             "config_data"
12         } else {
13             raise ConfigError("Config not found")
14         }
15     }
16
17     fn connect_database(config : String) -> String raise DatabaseError {
18         if config == "config_data" {
19             "connected"
20         } else {
21             raise DatabaseError("Invalid config")
22         }
23     }
24
25     fn initialize_app() -> String raise {
26         let config = load_config() catch {
27             ConfigError(msg) => raise Failure("Config error: " + msg)
28         }
29         let db = connect_database(config) catch {
30             DatabaseError(msg) => raise Failure("Database error: " + msg)
31         }
32         "App initialized with " + db
33     }
34
35     let app_result = try! initialize_app()
36     inspect(app_result, content="App initialized with connected")
37 }

```

Best Practices

- **Use specific error types:** Create custom suberror types for different error categories
- **Provide meaningful messages:** Include context and actionable information in error messages
- **Handle errors at appropriate levels:** Don't catch errors too early; let them propagate to where they can be properly handled
- **Use try!** for operations that should not fail: This will panic if an error occurs, making failures visible during development
- **Use try?** for recoverable errors: This returns a Result type that can be pattern matched
- **Implement proper cleanup:** Use the protect pattern or similar constructs for resource management

Performance Notes

- Error handling in MoonBit is zero-cost when no errors occur
- Error propagation is efficient and doesn't require heap allocation for the error path
- Custom error types with `derive(ToJson)` automatically generate efficient JSON serialization

HashSet

A mutable hash set based on a Robin Hood hash table.

Usage

Create

You can create an empty set using `new()` or construct it using `from_array()`.

```
1  ///|
2  test {
3      let _set1 = @hashset.of([1, 2, 3, 4, 5])
4      let _set2 : @hashset.HashSet[String] = @hashset.new()
5
6  }
```

Insert & Contain

You can use `insert()` to add a key to the set, and `contains()` to check whether a key exists.

```
1  ///|
2  test {
3      let set : @hashset.HashSet[String] = @hashset.new()
4      set.add("a")
5      assert_eq(set.contains("a"), true)
6  }
```

Remove

You can use `remove()` to remove a key.

```
1  ///|
2  test {
3      let set = @hashset.of(["a", "b", "c"])
4      set.remove("a")
5      assert_eq(set.contains("a"), false)
6  }
```

Size & Capacity

You can use `size()` to get the number of keys in the set, or `capacity()` to get the current capacity.

```
1  ///|
2  test {
3    let set = @hashset.of(["a", "b", "c"])
4    assert_eq(set.size(), 3)
5    assert_eq(set.capacity(), 8)
6  }
```

Similarly, you can use `is_empty()` to check whether the set is empty.

```
1  ///|
2  test {
3    let set : @hashset.HashSet[Int] = @hashset.new()
4    assert_eq(set.is_empty(), true)
5  }
```

Clear

You can use `clear` to remove all keys from the set, but the allocated memory will not change.

```
1  ///|
2  test {
3    let set = @hashset.of(["a", "b", "c"])
4    set.clear()
5    assert_eq(set.is_empty(), true)
6  }
```

Iteration

You can use `each()` or `eachi()` to iterate through all keys.

```
1  ///|
2  test {
3    let set = @hashset.of(["a", "b", "c"])
4    let arr = []
5    set.each(k => arr.push(k))
6    let arr2 = []
7    set.eachi((i, k) => arr2.push((i, k)))
8  }
```

Set Operations

You can use `union()`, `intersection()`, `difference()` and `symmetric_difference()` to perform set operations.

```

1  ///|
2  test {
3      let m1 = @hashset.of(["a", "b", "c"])
4      let m2 = @hashset.of(["b", "c", "d"])
5      fn to_sorted_array(set : @hashset.HashSet[String]) {
6          let arr = set.to_array()
7          arr.sort()
8          arr
9      }
10
11     assert_eq(m1.union(m2) |> to_sorted_array, ["a", "b", "c", "d"])
12     assert_eq(m1.intersection(m2) |> to_sorted_array, ["b", "c"])
13     assert_eq(m1.difference(m2) |> to_sorted_array, ["a"])
14     assert_eq(m1.symmetric_difference(m2) |> to_sorted_array, ["a", "d"])
15 }

```

Rational (DEPRECATED)

& ¶**This module is deprecated. Use @rational in module moonbitlang/x instead. Note that you need to rename Rational to Rational64.**

The Rational type represents a rational number, which is a number that can be expressed as a fraction a/b where a and b are integers and b is not zero.

All tests and examples have been removed. Please refer to the new moonbitlang/x module for updated documentation and examples.

bool

This package provides utility functions for working with boolean values in MoonBit, primarily focused on type conversions that are useful in systems programming, bitwise operations, and numerical computations.

Overview

Boolean values in MoonBit can be seamlessly converted to numeric types, following the standard convention where true maps to 1 and false maps to 0. This is particularly useful for:

- Conditional arithmetic and accumulation
- Interfacing with C libraries or low-level code
- Implementing boolean algebra with numeric operations
- Converting logical results to flags or indices

Basic Integer Conversion

Convert boolean values to standard integers for arithmetic operations:

```

1  ///|
2  test "bool to integer conversions" {
3    // Basic conversions
4    inspect(true.to_int(), content="1")
5    inspect(false.to_int(), content="0")
6
7    // Useful for conditional arithmetic
8    let score = 100
9    let bonus_applied = true
10   let final_score = score + bonus_applied.to_int() * 50
11   inspect(final_score, content="150")
12
13   // Accumulating boolean conditions
14   let conditions = [true, false, true, true, false]
15   let count = conditions.fold(init=0, fn(acc, cond) { acc + cond.to_int() })
16   inspect(count, content="3")
17 }

```

Specialized Integer Types

For specific use cases requiring different integer widths and signedness:

```

1  ///|
2  test "bool to specialized integer types" {
3    let flag = true
4    let no_flag = false
5
6    // UInt - useful for bit manipulation and flags
7    inspect(flag.to_uint(), content="1")
8    inspect(no_flag.to_uint(), content="0")
9
10   // Int64 - for large computations and compatibility
11   inspect(flag.to_int64(), content="1")
12   inspect(no_flag.to_int64(), content="0")
13
14   // UInt64 - for unsigned 64-bit operations
15   inspect(flag.to_uint64(), content="1")
16   inspect(no_flag.to_uint64(), content="0")
17 }

```

Practical Use Cases

Boolean Indexing and Selection

```

1  ///|
2  test "boolean indexing" {
3    // Use boolean conversion for array indexing
4    let options = ["default", "enhanced"]
5    let use_enhanced = true
6    let selected = options[use_enhanced.to_int()]
7    inspect(selected, content="enhanced")
8
9    // Conditional selection without branching
10   let base_value = 10
11   let multiplier = 2
12   let apply_multiplier = false
13   let result = base_value * (1 + apply_multiplier.to_int()) * (multiplier
14   inspect(result, content="10") // 10 * (1 + 0 * 1) = 10
15 }

```

Bit Manipulation and Flags

```

1  ///|
2  test "flags and bit operations" {
3    // Convert booleans to create bit flags
4    let read_permission = true
5    let write_permission = false
6    let execute_permission = true
7    let permissions = (read_permission.to_uint() << 2) |
8    (write_permission.to_uint() << 1) |
9    execute_permission.to_uint()
10   inspect(permissions, content="5") // Binary: 101 (read + execute)
11 }

```

Statistical and Mathematical Operations

```

1  ///|
2  test "statistical operations" {
3    // Calculate success rate from boolean results
4    let test_results = [true, true, false, true, false, true, true]
5    let successes = test_results.fold(init=0, fn(acc, result) {
6      acc + result.to_int()
7    })
8    let total = test_results.length()
9    let success_rate = successes.to_double() / total.to_double()
10   inspect(success_rate > 0.7, content="true")
11
12   // Boolean to numeric conversion for weighted calculations
13   let feature_enabled = [true, false, true]
14   let weights = [0.6, 0.3, 0.1]
15
16   // Calculate weighted score manually to avoid zip complexity
17   let score1 = feature_enabled[0].to_int().to_double() * weights[0]
18   let score2 = feature_enabled[1].to_int().to_double() * weights[1]
19   let score3 = feature_enabled[2].to_int().to_double() * weights[2]
20   let weighted_score = score1 + score2 + score3
21   inspect(weighted_score == 0.7, content="true")
22 }

```

This package provides the essential bridge between MoonBit's boolean logic and numeric computations, enabling elegant solutions for conditional arithmetic, flag operations, and data processing workflows.

Coverage Package Documentation

This package provides code coverage tracking utilities for MoonBit programs. It includes tools for measuring which parts of your code are executed during testing and generating coverage reports.

Coverage Counter

The core component for tracking code execution:

```
1  ///|
2  test "coverage counter basics" {
3    // Create a coverage counter for tracking 5 code points
4    let counter = CoverageCounter::new(5)
5
6    // Initially all counters should be zero
7    inspect(counter.to_string(), content="[0, 0, 0, 0, 0]")
8
9    // Increment specific tracking points
10   counter.incr(0) // First code point executed once
11   counter.incr(2) // Third code point executed once
12   counter.incr(0) // First code point executed again
13
14   // Check the updated counters
15   inspect(counter.to_string(), content="[2, 0, 1, 0, 0]")
16 }
```

Tracking Code Execution

Use coverage counters to track which code paths are executed:


```

1  ///|
2  test "tracking execution paths" {
3      let counter = CoverageCounter::new(3)
4      fn conditional_function(x : Int, coverage : CoverageCounter) -> String
5          if x > 0 {
6              coverage.incr(0) // Positive path
7              "positive"
8          } else if x < 0 {
9              coverage.incr(1) // Negative path
10             "negative"
11          } else {
12              coverage.incr(2) // Zero path
13              "zero"
14          }
15      }
16
17      // Test different paths
18      let result1 = conditional_function(5, counter)
19      inspect(result1, content="positive")
20      let result2 = conditional_function(-3, counter)
21      inspect(result2, content="negative")
22      let result3 = conditional_function(0, counter)
23      inspect(result3, content="zero")
24
25      // All paths should have been executed once
26      inspect(counter.to_string(), content="[1, 1, 1]")
27  }

```

Loop Coverage Tracking

Track coverage in loops and iterations:

```

1  ///|
2  test "loop coverage" {
3    let counter = CoverageCounter::new(2)
4    fn process_array(arr : Array[Int], coverage : CoverageCounter) -> Int
5      let mut sum = 0
6      for x in arr {
7        if x % 2 == 0 {
8          coverage.incr(0) // Even number processing
9          sum = sum + x
10         } else {
11           coverage.incr(1) // Odd number processing
12           sum = sum + x * 2
13         }
14       }
15       sum
16     }
17
18     let test_data = [1, 2, 3, 4, 5] // Mix of even and odd
19     let result = process_array(test_data, counter)
20
21     // Should have processed both even and odd numbers
22     inspect(result, content="24") // 1*2 + 2 + 3*2 + 4 + 5*2 = 2 + 2 + 6 +
23
24     // Both branches should have been executed
25     let coverage_str = counter.to_string()
26     inspect(coverage_str.length() > 5, content="true") // Should show execo
27   }

```

Function Coverage

Track coverage across different functions:

```

1  ///|
2  test "function coverage" {
3    let counter = CoverageCounter::new(4)
4    fn math_operations(
5      a : Int,
6      b : Int,
7      op : String,
8      coverage : CoverageCounter,
9    ) -> Int {
10     match op {
11       "add" => {
12         coverage.incr(0)
13         a + b
14       }
15       "sub" => {
16         coverage.incr(1)
17         a - b
18       }
19       "mul" => {
20         coverage.incr(2)
21         a * b
22       }
23       _ => {
24         coverage.incr(3)
25         0 // Unknown operation
26       }
27     }
28   }
29
30   // Test different operations
31   let add_result = math_operations(10, 5, "add", counter)
32   inspect(add_result, content="15")
33   let sub_result = math_operations(10, 5, "sub", counter)
34   inspect(sub_result, content="5")
35   let unknown_result = math_operations(10, 5, "unknown", counter)
36   inspect(unknown_result, content="0")
37
38   // Check that three of four branches were executed
39   let final_coverage = counter.to_string()
40   inspect(final_coverage, content="[1, 1, 0, 1]") // add, sub, not mul,
41 }

```

Coverage Analysis

Analyze coverage data to understand code execution:

```

1  ///|
2  test "coverage analysis" {
3      let counter = CoverageCounter::new(6)
4      fn complex_function(input : Int, coverage : CoverageCounter) -> String
5          coverage.incr(0) // Function entry
6          if input < 0 {
7              coverage.incr(1) // Negative branch
8              return "negative"
9          }
10         coverage.incr(2) // Non-negative path
11         if input == 0 {
12             coverage.incr(3) // Zero branch
13             return "zero"
14         }
15         coverage.incr(4) // Positive path
16         if input > 100 {
17             coverage.incr(5) // Large number branch
18             "large"
19         } else {
20             "small"
21         }
22     }
23
24     // Test various inputs
25     let result1 = complex_function(-5, counter)
26     inspect(result1, content="negative")
27     let result2 = complex_function(0, counter)
28     inspect(result2, content="zero")
29     let result3 = complex_function(50, counter)
30     inspect(result3, content="small")
31
32     // Analyze coverage: which paths were taken
33     let coverage = counter.to_string()
34     // Should show: [3, 1, 2, 1, 1, 0] - entry(3), negative(1), non-negati
35     inspect(coverage.length() > 10, content="true")
36 }

```

Integration with Testing

Coverage tracking integrates with MoonBit's testing system:

```

1  ///|
2  test "testing integration" {
3    // In real usage, coverage counters are typically generated automatica
4    // by the compiler for coverage analysis
5
6    fn test_function_with_coverage() -> Bool {
7      // This would normally have auto-generated coverage tracking
8      let counter = CoverageCounter::new(2)
9      fn helper(condition : Bool, cov : CoverageCounter) -> String {
10         if condition {
11             cov.incr(0)
12             "true_branch"
13         } else {
14             cov.incr(1)
15             "false_branch"
16         }
17     }
18
19     // Test both branches
20     let result1 = helper(true, counter)
21     let result2 = helper(false, counter)
22     result1 == "true_branch" && result2 == "false_branch"
23 }
24
25 let test_passed = test_function_with_coverage()
26 inspect(test_passed, content="true")
27 }

```

Coverage Reporting

Generate and analyze coverage reports:

```

1  ///|
2  test "coverage reporting" {
3    let counter = CoverageCounter::new(3)
4
5    // Simulate some code execution
6    counter.incr(0) // Line 1 executed
7    counter.incr(0) // Line 1 executed again
8    counter.incr(2) // Line 3 executed
9    // Line 2 (index 1) never executed
10
11    let report = counter.to_string()
12    inspect(report, content="[2, 0, 1]")
13
14    // In real usage, you might analyze this data:
15    fn analyze_coverage(_coverage_str : String) -> (Int, Int) {
16      // This would parse the coverage data and return (covered, total)
17      // For demonstration, we'll return mock values
18      (2, 3) // 2 out of 3 lines covered
19    }
20
21    let (covered, total) = analyze_coverage(report)
22    inspect(covered, content="2")
23    inspect(total, content="3")
24  }

```

Best Practices

1. Automatic Coverage Generation

In real applications, coverage tracking is typically generated automatically:

```

1  ///|
2  /// This is conceptual - actual coverage is compiler-generated
3  fn example_function(x : Int) -> String {
4    // Compiler automatically inserts: coverage.incr(0)
5    if x > 0 {
6      // Compiler automatically inserts: coverage.incr(1)
7      "positive"
8    } else {
9      // Compiler automatically inserts: coverage.incr(2)
10     "non-positive"
11   }
12   // Compiler automatically inserts: coverage.incr(3)
13 }
14
15 ///|
16 test "automatic coverage concept" {
17   let result = example_function(5)
18   inspect(result, content="positive")
19 }

```

2. Coverage-Driven Testing

Use coverage information to improve test quality:

```
1  ///|
2  test "coverage driven testing" {
3    // Write tests to ensure all code paths are covered
4    fn multi_branch_function(a : Int, b : Int) -> String {
5      if a > b {
6        "greater"
7      } else if a < b {
8        "less"
9      } else {
10       "equal"
11     }
12   }
13
14   // Test all branches
15   inspect(multi_branch_function(5, 3), content="greater")
16   inspect(multi_branch_function(2, 7), content="less")
17   inspect(multi_branch_function(4, 4), content="equal")
18
19   // This ensures 100% branch coverage
20 }
```

Integration with Build System

Coverage tracking integrates with MoonBit's build tools:

- Use moon test to run tests with coverage tracking
- Use moon coverage analyze to generate coverage reports
- Coverage data helps identify untested code paths
- Supports both line coverage and branch coverage analysis

Performance Considerations

- Coverage tracking adds minimal runtime overhead
- Counters use efficient fixed arrays for storage
- Coverage instrumentation is typically removed in release builds
- Use coverage data to optimize test suite performance

Common Use Cases

- **Test Quality Assessment:** Ensure comprehensive test coverage
- **Dead Code Detection:** Find unused code paths
- **Regression Testing:** Verify that tests exercise the same code paths
- **Performance Analysis:** Identify frequently executed code for optimization
- **Code Review:** Understand which parts of code are well-tested

The coverage package provides essential tools for maintaining high-quality, well-tested MoonBit code through comprehensive coverage analysis.

MoonBit Float Package Documentation

This package provides operations on 32-bit floating-point numbers (Float). It includes basic arithmetic, trigonometric functions, exponential and logarithmic functions, as well as utility functions for rounding and conversion.

Special Values

The package defines several special floating-point values:

```
1  ///|
2  test "special float values" {
3    // Infinity values
4    inspect(@float.infinity, content="Infinity")
5    inspect(@float.neg_infinity, content="-Infinity")
6
7    // Not a Number
8    inspect(@float.not_a_number, content="NaN")
9
10   // Bounds
11   inspect(@float.max_value, content="3.4028234663852886e+38")
12   inspect(@float.min_value, content="-3.4028234663852886e+38")
13   inspect(@float.min_positive, content="1.1754943508222875e-38")
14 }
15
16 ///|
17 test "checking special values" {
18   // Testing for special values
19   inspect(@float.infinity.is_inf(), content="true")
20   inspect(@float.neg_infinity.is_neg_inf(), content="true")
21   inspect(@float.infinity.is_pos_inf(), content="true")
22   inspect(@float.not_a_number.is_nan(), content="true")
23 }
```

Rounding Functions

The package provides various ways to round floating-point numbers:


```

1  ///|
2  test "rounding functions" {
3    // Ceiling - rounds up
4    inspect(@float.ceil(3.2), content="4")
5    inspect(@float.ceil(-3.2), content="-3")
6
7    // Floor - rounds down
8    inspect(@float.floor(3.2), content="3")
9    inspect(@float.floor(-3.2), content="-4")
10
11   // Round - rounds to nearest integer
12   inspect(@float.round(3.7), content="4")
13   inspect(@float.round(3.2), content="3")
14
15   // Truncate - removes decimal part
16   inspect(@float.trunc(3.7), content="3")
17   inspect(@float.trunc(-3.7), content="-3")
18 }

```

Utility Functions

Other useful operations on floats:

```

1  ///|
2  test "utility functions" {
3    // Absolute value
4    inspect(@float.abs(-3.14), content="3.140000104904175")
5
6    // Conversion to integer
7    inspect(3.14.to_int(), content="3")
8
9    // Default value
10   inspect(@float.default(), content="0")
11 }

```

Byte Representation

Functions to convert floats to their byte representation:

```

1  ///|
2  test "byte representation" {
3    let x : Float = 3.14
4    // Big-endian bytes
5    let be_bytes = x.to_be_bytes()
6    // Little-endian bytes
7    let le_bytes = x.to_le_bytes()
8    inspect(be_bytes.length(), content="4")
9    inspect(le_bytes.length(), content="4")
10 }

```

Method Style

All functions can also be called in method style:

```

1  ///|
2  test "method style calls" {
3    let x : Float = 3.14
4    inspect(x.floor(), content="3")
5    inspect(x.ceil(), content="4")
6    inspect(x.round(), content="3")
7    let y : Float = 2.0
8    inspect(y.pow(3.0), content="8")
9  }

```

int16

This package provides a fixed-width 16-bit signed integer type.

Range and Constants

The Int16 type represents values from -32768 to 32767 (inclusive). The package provides these boundary values as constants:

```

1  ///|
2  test "int16 range" {
3    inspect(@int16.min_value, content="-32768")
4    inspect(@int16.max_value, content="32767")
5  }

```

Arithmetic Operations

The Int16 type supports standard arithmetic operations:

```

1  ///|
2  test "int16 arithmetic" {
3    let a : Int16 = 100
4    let b : Int16 = 50
5
6    // Basic arithmetic
7    inspect(a + b, content="150")
8    inspect(a - b, content="50")
9    inspect(a * b, content="5000")
10   inspect(a / b, content="2")
11
12   // Overflow behavior
13   let max = @int16.max_value
14   let min = @int16.min_value
15   inspect(max + 1, content="-32768") // Wraps around to min_value
16   inspect(min - 1, content="32767") // Wraps around to max_value
17 }

```

Bitwise Operations

Int16 supports standard bitwise operations:

```

1  ///|
2  test "int16 bitwise" {
3    let a : Int16 = 0b1100
4    let b : Int16 = 0b1010
5
6    // Bitwise AND, OR, XOR
7    inspect(a & b, content="8") // 0b1000
8    inspect(a | b, content="14") // 0b1110
9    inspect(a ^ b, content="6") // 0b0110
10
11   // Bit shifts
12   let x : Int16 = 8
13   inspect(x << 1, content="16") // Left shift
14   inspect(x >> 1, content="4") // Right shift
15 }

```

Comparison Operations

Int16 implements the Compare trait for total ordering:

```

1  ///|
2  test "int16 comparison" {
3    let a : Int16 = 100
4    let b : Int16 = 50
5    let c : Int16 = 100
6
7    // Equality
8    inspect(a == b, content="false")
9    inspect(a == c, content="true")
10
11   // Ordering
12   inspect(a > b, content="true")
13   inspect(b < c, content="true")
14
15   // Compare function returns -1, 0, or 1
16   inspect(a.compare(b), content="1")
17   inspect(b.compare(c), content="-1")
18   inspect(a.compare(c), content="0")
19 }

```

Default Value

Int16 implements the Default trait, with 0 as its default value:

```

1  ///|
2  test "int16 default" {
3    let x = Int16::default()
4    inspect(x, content="0")
5  }

```

Type Coercion and Conversion

Integer literals can be coerced to Int16 when the type is explicitly specified:

```

1  ///|
2  test "int16 coercion" {
3    let a : Int16 = 42 // Coercion from integer literal
4    let b : Int16 = 0xFF // Hexadecimal literal
5    let c : Int16 = 0b1111 // Binary literal
6    inspect(a, content="42")
7    inspect(b, content="255")
8    inspect(c, content="15")
9  }

```

Set Package Documentation

This package provides a hash-based set data structure that maintains insertion order. The `Set[K]` type stores unique elements and provides efficient membership testing, insertion, and deletion operations.

Creating Sets

There are several ways to create sets:

```

1  ///|
2  test "creating sets" {
3    // Empty set
4    let empty_set : @set.Set[Int] = @set.Set::new()
5    inspect(empty_set.size(), content="0")
6    inspect(empty_set.is_empty(), content="true")
7
8    // Set with initial capacity
9    let set_with_capacity : @set.Set[Int] = @set.Set::new(capacity=16)
10   inspect(set_with_capacity.capacity(), content="16")
11
12   // From array
13   let from_array = @set.Set::from_array([1, 2, 3, 2, 1]) // Duplicates are removed
14   inspect(from_array.size(), content="3")
15
16   // From fixed array
17   let from_fixed = @set.Set::of([10, 20, 30])
18   inspect(from_fixed.size(), content="3")
19
20   // From iterator
21   let from_iter = @set.Set::from_iter([1, 2, 3, 4, 5].iter())
22   inspect(from_iter.size(), content="5")
23 }

```

Basic Operations

Add, remove, and check membership:

```

1  ///|
2  test "basic operations" {
3    let set = @set.Set::new()
4
5    // Adding elements
6    set.add("apple")
7    set.add("banana")
8    set.add("cherry")
9    inspect(set.size(), content="3")
10
11   // Adding duplicate (no effect)
12   set.add("apple")
13   inspect(set.size(), content="3") // Still 3
14
15   // Check membership
16   inspect(set.contains("apple"), content="true")
17   inspect(set.contains("orange"), content="false")
18
19   // Remove elements
20   set.remove("banana")
21   inspect(set.contains("banana"), content="false")
22   inspect(set.size(), content="2")
23
24   // Check if addition/removal was successful
25   let was_added = set.add_and_check("date")
26   inspect(was_added, content="true")
27   let was_added_again = set.add_and_check("date")
28   inspect(was_added_again, content="false") // Already exists
29   let was_removed = set.remove_and_check("cherry")
30   inspect(was_removed, content="true")
31   let was_removed_again = set.remove_and_check("cherry")
32   inspect(was_removed_again, content="false") // Doesn't exist
33 }

```

Set Operations

Perform mathematical set operations:

```

1  ///|
2  test "set operations" {
3      let set1 = @set.Set::from_array([1, 2, 3, 4])
4      let set2 = @set.Set::from_array([3, 4, 5, 6])
5
6      // Union (all elements from both sets)
7      let union_set = set1.union(set2)
8      let union_array = union_set.to_array()
9      inspect(union_array.length(), content="6") // [1, 2, 3, 4, 5, 6]
10
11     // Alternative union syntax
12     let union_alt = set1 | set2
13     inspect(union_alt.size(), content="6")
14
15     // Intersection (common elements)
16     let intersection_set = set1.intersection(set2)
17     let intersection_array = intersection_set.to_array()
18     inspect(intersection_array.length(), content="2") // [3, 4]
19
20     // Alternative intersection syntax
21     let intersection_alt = set1 & set2
22     inspect(intersection_alt.size(), content="2")
23
24     // Difference (elements in first but not second)
25     let difference_set = set1.difference(set2)
26     let difference_array = difference_set.to_array()
27     inspect(difference_array.length(), content="2") // [1, 2]
28
29     // Alternative difference syntax
30     let difference_alt = set1 - set2
31     inspect(difference_alt.size(), content="2")
32
33     // Symmetric difference (elements in either but not both)
34     let sym_diff_set = set1.symmetric_difference(set2)
35     let sym_diff_array = sym_diff_set.to_array()
36     inspect(sym_diff_array.length(), content="4") // [1, 2, 5, 6]
37
38     // Alternative symmetric difference syntax
39     let sym_diff_alt = set1 ^ set2
40     inspect(sym_diff_alt.size(), content="4")
41 }

```

Set Relationships

Test relationships between sets:

```

1  ///|
2  test "set relationships" {
3    let small_set = @set.Set::from_array([1, 2])
4    let large_set = @set.Set::from_array([1, 2, 3, 4])
5    let disjoint_set = @set.Set::from_array([5, 6, 7])
6
7    // Subset testing
8    inspect(small_set.is_subset(large_set), content="true")
9    inspect(large_set.is_subset(small_set), content="false")
10
11   // Superset testing
12   inspect(large_set.is_superset(small_set), content="true")
13   inspect(small_set.is_superset(large_set), content="false")
14
15   // Disjoint testing (no common elements)
16   inspect(small_set.is_disjoint(disjoint_set), content="true")
17   inspect(small_set.is_disjoint(large_set), content="false")
18
19   // Equal sets
20   let set1 = @set.Set::from_array([1, 2, 3])
21   let set2 = @set.Set::from_array([3, 2, 1]) // Order doesn't matter
22   inspect(set1 == set2, content="true")
23 }

```

Iteration and Conversion

Iterate over sets and convert to other types:

```

1  ///|
2  test "iteration and conversion" {
3    let set = @set.Set::from_array(["first", "second", "third"])
4
5    // Convert to array (maintains insertion order)
6    let array = set.to_array()
7    inspect(array.length(), content="3")
8
9    // Iterate over elements
10   let mut count = 0
11   set.each(fn(_element) { count = count + 1 })
12   inspect(count, content="3")
13
14   // Iterate with index
15   let mut indices_sum = 0
16   set.eachi(fn(i, _element) { indices_sum = indices_sum + i })
17   inspect(indices_sum, content="3") // 0 + 1 + 2 = 3
18
19   // Use iterator
20   let elements = set.iter().collect()
21   inspect(elements.length(), content="3")
22
23   // Copy a set
24   let copied_set = set.copy()
25   inspect(copied_set.size(), content="3")
26   inspect(copied_set == set, content="true")
27 }

```

Modifying Sets

Clear and modify existing sets:

```

1  ///|
2  test "modifying sets" {
3    let set = @set.Set::from_array([10, 20, 30, 40, 50])
4    inspect(set.size(), content="5")
5
6    // Clear all elements
7    set.clear()
8    inspect(set.size(), content="0")
9    inspect(set.is_empty(), content="true")
10
11   // Add elements back
12   set.add(100)
13   set.add(200)
14   inspect(set.size(), content="2")
15   inspect(set.contains(100), content="true")
16 }

```

JSON Serialization

Sets can be serialized to JSON as arrays:


```

1  ///|
2  test "json serialization" {
3      let set = @set.Set::from_array([1, 2, 3])
4      let json = set.to_json()
5
6      // JSON representation is an array
7      inspect(json, content="Array([Number(1), Number(2), Number(3)])")
8
9      // String set
10     let string_set = @set.Set::from_array(["a", "b", "c"])
11     let string_json = string_set.to_json()
12     inspect(
13         string_json,
14         content="Array([String(\"a\"), String(\"b\"), String(\"c\")])",
15     )
16 }

```

Working with Different Types

Sets work with any type that implements Hash and Eq:

```

1  ///|
2  test "different types" {
3      // Integer set
4      let int_set = @set.Set::from_array([1, 2, 3, 4, 5])
5      inspect(int_set.contains(3), content="true")
6
7      // String set
8      let string_set = @set.Set::from_array(["hello", "world", "moonbit"])
9      inspect(string_set.contains("world"), content="true")
10
11     // Note: Char and Bool types don't implement Hash in this version
12     // So we use Int codes for demonstration
13     let char_codes = @set.Set::from_array([97, 98, 99]) // ASCII codes for 'a', 'b', 'c'
14     inspect(char_codes.contains(98), content="true") // 'b' = 98
15
16     // Integer set representing boolean values
17     let bool_codes = @set.Set::from_array([1, 0, 1]) // 1=true, 0=false
18     inspect(bool_codes.size(), content="2") // Only 1 and 0
19 }

```

Performance Examples

Demonstrate efficient operations:

```

1  ///|
2  test "performance examples" {
3    // Large set operations
4    let large_set = @set.Set::new(capacity=1000)
5
6    // Add many elements
7    for i in 0..<100 {
8      large_set.add(i)
9    }
10   inspect(large_set.size(), content="100")
11
12   // Fast membership testing
13   inspect(large_set.contains(50), content="true")
14   inspect(large_set.contains(150), content="false")
15
16   // Efficient set operations on large sets
17   let another_set = @set.Set::new()
18   for i in 50..<150 {
19     another_set.add(i)
20   }
21   let intersection = large_set.intersection(another_set)
22   inspect(intersection.size(), content="50") // Elements 50-99
23 }

```

Use Cases

Sets are particularly useful for:

- **Removing duplicates:** Convert arrays to sets and back to remove duplicates
- **Membership testing:** Fast $O(1)$ average-case lookups
- **Mathematical operations:** Union, intersection, difference operations
- **Unique collections:** Maintaining collections of unique items
- **Algorithm implementation:** Graph algorithms, caching, etc.

Performance Characteristics

- **Insertion:** $O(1)$ average case, $O(n)$ worst case
- **Removal:** $O(1)$ average case, $O(n)$ worst case
- **Lookup:** $O(1)$ average case, $O(n)$ worst case
- **Space complexity:** $O(n)$ where n is the number of elements
- **Iteration order:** Maintains insertion order (linked hash set)

Best Practices

- **Pre-size when possible:** Use `@set.Set::new(capacity=n)` if you know the approximate size
- **Use appropriate types:** Ensure your key type has good Hash and Eq implementations
- **Prefer set operations:** Use built-in union, intersection, etc. instead of manual loops
- **Check return values:** Use `add_and_check` and `remove_and_check` when you need to know if the operation succeeded
- **Consider memory usage:** Sets have overhead compared to arrays for small collections

buffer

The buffer package provides a flexible byte buffer implementation for efficient binary data handling and serialization.

Basic Usage

Create a new buffer and write basic data:

```

1  ///|
2  test "basic buffer operations" {
3    let buf = @buffer.new()
4
5    // Write some bytes
6    buf..write_byte(b'H')..write_byte(b'i')
7
8    // Check contents
9    inspect(buf.is_empty(), content="false")
10   inspect(buf.length(), content="2")
11
12   // Get contents as bytes
13   let bytes = buf.contents()
14   inspect(
15     bytes,
16     content=(
17       #|b"\x48\x69"
18     ),
19   )
20
21   // Reset buffer
22   buf.reset()
23   inspect(buf.is_empty(), content="true")
24 }
```

Writing Numbers

Write numbers in different encodings:

```

1  ///|
2  test "number serialization" {
3      inspect(
4          @buffer.new( )
5          // Write integers in different byte orders
6          ..write_int_be(42)
7          ..write_int_le(42)
8          .to_bytes(),
9          content=(
10             #|b"\x00\x00\x00\x2a\x2a\x00\x00\x00"
11         ),
12     )
13     inspect(
14         @buffer.new( )
15         // Write floating point numbers
16         ..write_float_be(3.14)
17         ..write_float_le(3.14)
18         .to_bytes(),
19         content=(
20             #|b"\x40\x48\xf5\xc3\xc3\xf5\x48\x40"
21         ),
22     )
23     inspect(
24         @buffer.new( )
25         // Write 64-bit integers
26         ..write_int64_be(0xAABBCCDDEEL)
27         ..write_int64_le(0xAABBCCDDEEL)
28         .to_bytes(),
29         content=(
30             #|b"\x00\x00\x00\xaa\xbb\xcc\xdd\xee\xee\xdd\xcc\xbb\xaa\x00\x00\x00"
31         ),
32     )
33     inspect(
34         @buffer.new( )
35         // Write unsigned integers
36         ..write_uint_be(0x2077U)
37         ..write_uint_le(0x2077U)
38         .to_bytes(),
39         content=(
40             #|b"\x00\x00\x20\x77\x77\x20\x00\x00"
41         ),
42     )
43 }

```

Writing Byte Sequences

Write sequences of bytes:

```

1  ///|
2  test "byte sequence writing" {
3      let buf = @buffer.new()
4
5      // Write byte array
6      let bytes = b"Hello"
7      buf.write_bytes(bytes)
8
9      // Write byte iterator
10     buf.write_iter(bytes.iter())
11     let contents = buf.to_bytes()
12     inspect(
13         contents,
14         content=(
15             #|b"\x48\x65\x6c\x6c\x6f\x48\x65\x6c\x6c\x6f"
16         ),
17     ) // "Hello" written twice
18 }

```

Writing Structured Data

Write structured data that implements Show:

```

1  ///|
2  test "object writing" {
3      let buf = @buffer.new()
4
5      // Write int as object
6      buf.write_object(42)
7
8      // Contents will be "42" as bytes
9      let contents = buf.contents()
10     inspect(
11         contents,
12         content=(
13             #|b"\x34\x00\x32\x00"
14         ),
15     )
16 }

```

Size Hints

Provide size hints for better performance:

```

1  ///|
2  test "buffer with size hint" {
3    // Create buffer with initial capacity hint
4    let buf = @buffer.new(size_hint=1024)
5
6    // Write some data
7    for i in 0..<100 {
8      buf.write_int_le(i)
9    }
10
11    // Each integer takes 4 bytes
12    inspect(buf.length(), content="400")
13  }

```

Buffer as Logger

The buffer implements the Logger trait for Show:

```

1  ///|
2  test "buffer as logger" {
3    let buf = @buffer.new()
4    let array = [1, 2, 3]
5
6    // Use buffer to log array
7    array.output(buf)
8    let contents = buf.contents()
9    inspect(
10      contents,
11      content=(
12        #|b"\x5b\x00\x31\x00\x2c\x00\x20\x00\x32\x00\x2c\x00\x20\x00\x33\x00"
13      ),
14    )
15  }

```

Converting to String/Bytes

Methods for converting buffer contents:

```

1  ///|
2  test "buffer conversion" {
3    let buf = @buffer.new()
4    buf.write_byte(b'a')
5    buf.write_byte(b'b')
6    buf.write_byte(b'c')
7    let bytes = buf.to_bytes()
8    inspect(
9      bytes,
10     content=(
11       #|b"\x61\x62\x63"
12     ),
13   )
14 }

```

Binary Viewing

Support for viewing subsets of bytes:

```
1  ///|
2  test "byte view writing" {
3      let buf = @buffer.new()
4      let bytes = b"Hello World"
5
6      // Write a view of the bytes
7      buf.write_bytesview(bytes[0:5]) // Write "Hello"
8      let contents = buf.to_bytes()
9      inspect(
10         contents,
11         content=(
12             #|b"\x48\x65\x6c\x6c\x6f"
13         ),
14     )
15 }
```

Random

This is an efficient random number generation function based on the paper Fast Random Integer Generation in an Interval by Daniel Lemire, as well as the Golang's rand/v2 package.

Internally, it uses the ChaCha8 cipher to generate random numbers. It is a cryptographically secure pseudo-random number generator (CSPRNG) that is also very fast.

Usage

```

1  ///|
2  test {
3    let r = @random.Rand::new()
4    assert_eq(r.uint(limit=10), 7)
5    assert_eq(r.uint(limit=10), 0)
6    assert_eq(r.uint(limit=10), 5)
7    assert_eq(r.int(), 1064320769)
8    assert_eq(r.double(), 0.3318940049218405)
9    assert_eq(r.int(limit=10), 0)
10   assert_eq(r.uint(), 311122750)
11   assert_eq(r.int64(), 2043189202271773519)
12   assert_eq(r.int64(limit=10), 8)
13   assert_eq(r.uint64(), 3951155890335085418)
14   let a = [1, 2, 3, 4, 5]
15   r.shuffle(a.length(), (i, j) => {
16     let t = a[i]
17     a[i] = a[j]
18     a[j] = t
19   })
20   assert_eq(a, [2, 1, 4, 3, 5])
21 }

```

Strconv

This package implements conversions to and from string representations of basic data types.

Usage

Parse

Use `parse_bool`, `parse_double`, `parse_int`, and `parse_int64` convert strings to values.

```

1  ///|
2  test {
3    let b = @strconv.parse_bool("true")
4    assert_eq(b, true)
5    let i1 = @strconv.parse_int("1234567")
6    assert_eq(i1, 1234567)
7    let i2 = @strconv.parse_int("101", base=2)
8    assert_eq(i2, 5)
9    let d = @strconv.parse_double("123.4567")
10   assert_eq(d, 123.4567)
11 }

```

For types that implement the `FromStr` trait, you can also use helper function `parse` to convert a string to a value.


```

1  ///|
2  test {
3      let a : Int = @strconv.parse("123")
4      assert_eq(a, 123)
5      let b : Bool = @strconv.parse("true")
6      assert_eq(b, true)
7  }

```

Option

The Option type is a built-in type in MoonBit that represents an optional value. The type annotation Option[A] can also be written as A?.

It is an enum with two variants: Some(T), which represents a value of type T, and None, representing no value.

Note that some methods of the Option are defined in the core/builtin package.

Usage

Create

You can create an Option value using the Some and None constructors, remember to give proper type annotations.

```

1  ///|
2  test {
3      let some : Int? = Some(42)
4      let none : String? = None
5      inspect(some, content="Some(42)")
6      inspect(none, content="None")
7  }

```

Extracting values

You can extract the value from an Option using the match expression (Pattern Matching).

```

1  ///|
2  test {
3      let i = Some(42)
4      let j = match i {
5          Some(value) => value
6          None => abort("unreachable")
7      }
8      assert_eq(j, 42)
9  }

```

Or using the unwrap method, which will panic if the result is None and return the value if it is Some.

```

1  ///|
2  test {
3      let some : Int? = Some(42)
4      let value = some.unwrap() // 42
5      assert_eq(value, 42)
6  }

```

A safer alternative to `unwrap` is the `or` method, which returns the value if it is `Some`, otherwise, it returns the default value.

```

1  ///|
2  test {
3      let none : Int? = None
4      let value = none.unwrap_or(0) // 0
5      assert_eq(value, 0)
6  }

```

There is also the `or_else` method, which returns the value if it is `Some`, otherwise, it returns the result of the provided function.

```

1  ///|
2  test {
3      let none : Int? = None
4      let value = none.unwrap_or_else(() => 0) // 0
5      assert_eq(value, 0)
6  }

```

Transforming values

You can transform the value of an `Option` using the `map` method. It applies the provided function to the value if it is `Some`, otherwise, it returns `None`.

```

1  ///|
2  test {
3      let some : Int? = Some(42)
4      let new_some = some.map((value : Int) => value + 1) // Some(43)
5      assert_eq(new_some, Some(43))
6  }

```

There is a `filter` method that applies a predicate to the value if it is `Some`, otherwise, it returns `None`.

```

1  ///|
2  test {
3      let some : Int? = Some(42)
4      let new_some = some.filter((value : Int) => value > 40) // Some(42)
5      let none = some.filter((value : Int) => value > 50) // None
6      assert_eq(new_some, Some(42))
7      assert_eq(none, None)
8  }

```

Monadic operations

You can chain multiple operations that return `Option` using the `bind` method, which applies a function to the value if it is `Some`, otherwise, it returns `None`. Different from `map`, the function in argument returns an `Option`.

```
1  ///|
2  test {
3    let some : Int? = Some(42)
4    let new_some = some.bind((value : Int) => Some(value + 1)) // Some(43)
5    assert_eq(new_some, Some(43))
6  }
```

Sometimes we want to reduce the nested `Option` values into a single `Option`, you can use the `flatten` method to achieve this. It transforms `Some(Some(value))` into `Some(value)`, and `None` otherwise.

```
1  ///|
2  test {
3    let some : Int?? = Some(Some(42))
4    let new_some = some.flatten() // Some(42)
5    assert_eq(new_some, Some(42))
6    let none : Int?? = Some(None)
7    let new_none = none.flatten() // None
8    assert_eq(new_none, None)
9  }
```

uint64

The `moonbitlang/core/uint64` package provides functionality for working with 64-bit unsigned integers. This package includes constants, operators, and conversions for `UInt64` values.

Constants

The package defines the minimum and maximum values for `UInt64`:

```
1  ///|
2  test "UInt64 constants" {
3    // Minimum value of UInt64
4    inspect(@uint64.min_value, content="0")
5
6    // Maximum value of UInt64
7    inspect(@uint64.max_value, content="18446744073709551615")
8  }
```

Arithmetic Operations

`UInt64` supports standard arithmetic operations:

```

1  ///|
2  test "UInt64 arithmetic" {
3      let a : UInt64 = 100UL
4      let b : UInt64 = 50UL
5
6      // Addition
7      inspect(a + b, content="150")
8
9      // Subtraction
10     inspect(a - b, content="50")
11
12     // Multiplication
13     inspect(a * b, content="5000")
14
15     // Division
16     inspect(a / b, content="2")
17
18     // Overflow behavior
19     inspect(@uint64.max_value + 1UL, content="0") // Wraps around to 0
20     inspect(@uint64.min_value - 1UL, content="18446744073709551615") // Un
21 }

```

Bitwise Operations

UInt64 supports various bitwise operations:

```

1  ///|
2  test "UInt64 bitwise operations" {
3      let a : UInt64 = 0b1010UL
4      let b : UInt64 = 0b1100UL
5
6      // Bitwise AND
7      inspect(a & b, content="8")
8
9      // Bitwise OR
10     inspect(a | b, content="14")
11
12     // Bitwise XOR
13     inspect(a ^ b, content="6")
14
15     // Left shift
16     inspect(a << 1, content="20")
17     inspect(a << 2, content="40")
18
19     // Right shift
20     inspect(a >> 1, content="5")
21     inspect(b >> 2, content="3")
22 }

```

Comparison and Equality

UInt64 supports comparison and equality operations:

```

1  ///|
2  test "UInt64 comparison and equality" {
3      let a : UInt64 = 100UL
4      let b : UInt64 = 50UL
5      let c : UInt64 = 100UL
6
7      // Equality
8      inspect(a == c, content="true")
9      inspect(a != b, content="true")
10
11     // Comparison
12     inspect(a > b, content="true")
13     inspect(b < a, content="true")
14     inspect(a >= c, content="true")
15     inspect(c <= a, content="true")
16 }

```

Byte Conversion

UInt64 provides methods for converting to bytes in both big-endian and little-endian formats:

```

1  ///|
2  test "UInt64 byte conversion" {
3      // Convert to bytes in big-endian order (most significant byte first)
4      let be_bytes = 0x123456789ABCDEF0UL.to_be_bytes()
5      inspect(
6          be_bytes,
7          content=(
8              #|b"\x12\x34\x56\x78\x9a\xbc\xde\xfo"
9          ),
10     )
11
12     // Convert to bytes in little-endian order (least significant byte first)
13     let le_bytes = 0x123456789ABCDEF0UL.to_le_bytes()
14     inspect(
15         le_bytes,
16         content=(
17             #|b"\xf0\xde\xbc\x9a\x78\x56\x34\x12"
18         ),
19     )
20 }

```

Default Value and Hashing

UInt64 implements the Default trait:

```

1  ///|
2  test "UInt64 default value" {
3    // Default value is 0
4    let a : UInt64 = 0UL
5    inspect(a, content="0")
6
7    // Hash support is available via .hash()
8    let value : UInt64 = 42UL
9    inspect(value.hash(), content="-1962516083")
10 }

```

Type Conversions

UInt64 works with various conversions to and from other types:

```

1  ///|
2  test "UInt64 conversions" {
3    // From Int to UInt64
4    inspect((42).to_uint64(), content="42")
5
6    // From UInt64 to Int or Double
7    let value : UInt64 = 100UL
8    inspect(value.to_int(), content="100")
9    let as_double = value.to_double()
10   inspect(as_double, content="100")
11
12   // Overflow handling in conversions
13   inspect((-1).to_uint64(), content="18446744073709551615") // Negative
14
15   // Converting back from floating point
16   let from_double = 42.0.to_uint64()
17   inspect(from_double, content="42")
18 }

```

Working with Large Numbers

UInt64 is especially useful for applications requiring large unsigned integers:

```

1  ///|
2  test "UInt64 for large numbers" {
3    // UInt64 can represent very large numbers
4    let large_number : UInt64 = (1UL << 63) - 1UL
5
6    // This exceeds a 32-bit integer's maximum value
7    inspect(large_number > (1UL << 32) - 1UL, content="true")
8
9    // Arithmetic still works with large values
10   let result = large_number * 2UL
11   inspect(result, content="18446744073709551614") // This effectively ca
12 }

```

Working with Hexadecimal Literals

UInt64 works well with hexadecimal literals for clarity when working with bit patterns:

```
1  ///|
2  test "UInt64 hexadecimal literals" {
3    // Using hex literals for better readability when working with bit patterns
4    let value = 0xDEADBEEFUL
5
6    // Extract specific byte using shifts and masks
7    let ad = (value >> 16) & 0xFFUL
8    inspect(ad.to_byte(), content="b'\xAD'")
9
10   // Convert to byte representation
11   let bytes = value.to_be_bytes()
12   inspect(
13     bytes,
14     content=(
15       #|b"\x00\x00\x00\x00\xde\xad\xbe\xef"
16     ),
17   )
18 }
```

Queue

Queue is a first in first out (FIFO) data structure, allowing to process their elements in the order they come.

Usage

Create and Clear

You can create a queue manually by using the new or construct it using the from_array.

```
1  ///|
2  test {
3    let _queue : @queue.Queue[Int] = @queue.new()
4    let _queue1 = @queue.of([1, 2, 3])
5
6  }
```

To clear the queue, you can use the clear method.

```
1  ///|
2  test {
3    let queue = @queue.of([1, 2, 3])
4    queue.clear()
5  }
```

Length

You can get the length of the queue by using the length method. The is_empty method can be used to check if the queue is empty.

```

1  ///|
2  test {
3    let queue = @queue.of([1, 2, 3])
4    assert_eq(queue.length(), 3)
5    assert_eq(queue.is_empty(), false)
6  }

```

Pop and Push

You can add elements to the queue using the push method and remove them using the pop method.

```

1  ///|
2  test {
3    let queue = @queue.new()
4    queue.push(1)
5    queue.push(2)
6    assert_eq(queue.pop(), Some(1))
7    assert_eq(queue.pop(), Some(2))
8  }

```

Peek

You can get the first element of the queue without removing it using the peek method.

```

1  ///|
2  test {
3    let queue = @queue.of([1, 2, 3])
4    assert_eq(queue.peek(), Some(1))
5  }

```

Traverse

You can traverse the queue using the each method.

```

1  ///|
2  test {
3    let queue = @queue.of([1, 2, 3])
4    let mut sum = 0
5    queue.each(x => sum += x)
6    assert_eq(sum, 6)
7  }

```

You can fold the queue using the fold method.

```

1  ///|
2  test {
3    let queue = @queue.of([1, 2, 3])
4    let sum = queue.fold(init=0, (acc, x) => acc + x)
5    assert_eq(sum, 6)
6  }

```

Copy and Transfer

You can copy a queue using the copy method.


```

1  ///|
2  test {
3      let queue = @queue.of([1, 2, 3])
4      let _queue2 = queue.copy()
5
6  }

```

Transfer the elements from one queue to another using the transfer method.

```

1  ///|
2  test {
3      let dst : @queue.Queue[Int] = @queue.new()
4      let src : @queue.Queue[Int] = @queue.of([5, 6, 7, 8])
5      src.transfer(dst)
6  }

```

double

This package provides comprehensive support for double-precision floating-point arithmetic, including basic operations, trigonometric functions, exponential and logarithmic functions, as well as utility functions for handling special values

.

Constants and Special Values

The package provides several important constants and special floating-point values:

```

1  ///|
2  test "special values" {
3      // Special values
4      inspect(@double.infinity, content="Infinity")
5      inspect(@double.neg_infinity, content="-Infinity")
6      inspect(@double.not_a_number, content="NaN")
7
8      // Limits
9      inspect(@double.max_value, content="1.7976931348623157e+308")
10     inspect(@double.min_value, content="-1.7976931348623157e+308")
11     inspect(@double.min_positive, content="2.2250738585072014e-308")
12 }

```

Basic Operations

Basic mathematical operations and rounding functions:

```

1  ///|
2  test "basic operations" {
3    // Absolute value
4    inspect(@double.abs(-3.14), content="3.14")
5
6    // Rounding functions
7    inspect(@double.floor(3.7), content="3")
8    inspect(@double.ceil(3.2), content="4")
9    inspect(@double.round(3.5), content="4")
10   inspect(@double.trunc(3.7), content="3")
11
12   // Pow function
13   inspect(2.0.pow(3), content="8")
14
15   // Sign
16   inspect((-3.14).signum(), content="-1")
17   inspect(2.0.signum(), content="1")
18
19   // Type conversion
20   inspect(@double.from_int(42), content="42")
21 }

```

Special Value Testing

Functions for testing special floating-point values and comparing numbers:

```

1  ///|
2  test "special value testing" {
3    // Testing for special values
4    inspect(@double.not_a_number.is_nan(), content="true")
5    inspect(@double.infinity.is_inf(), content="true")
6    inspect(@double.infinity.is_pos_inf(), content="true")
7    inspect(@double.neg_infinity.is_neg_inf(), content="true")
8
9    // Approximate equality
10   let relative_tolerance = 1.e-9
11   inspect(@double.is_close(0.1 + 0.2, 0.3, relative_tolerance~), content="true")
12 }

```

Binary Representation

Functions for converting doubles to their binary representation:

```

1  ///|
2  test "binary representation" {
3    let num = 1.0
4
5    // Convert to big-endian and little-endian bytes
6    // Different byte orders should produce different results
7    inspect(
8      num.to_be_bytes(),
9      content=(
10       #|b"\x3f\xf0\x00\x00\x00\x00\x00\x00"
11     ),
12   )
13   inspect(
14     num.to_le_bytes(),
15     content=(
16       #|b"\x00\x00\x00\x00\x00\x00\xf0\x3f"
17     ),
18   )
19 }

```

Note: Most methods can be called either as a method (`d.to_be_bytes()`) or as a package function (`@double.to_be_bytes(d)`).

Sorted Map

A mutable map backed by an AVL tree that maintains keys in sorted order.

Overview

SortedMap is an ordered map implementation that keeps entries sorted by keys. It provides efficient lookup, insertion, and deletion operations, with stable traversal order based on key comparison.

Performance

- **add/set**: $O(\log n)$
- **remove**: $O(\log n)$
- **get/contains**: $O(\log n)$
- **iterate**: $O(n)$
- **range**: $O(\log n + k)$ where k is number of elements in range
- **space complexity**: $O(n)$

Usage

Create

You can create an empty SortedMap or a SortedMap from other containers.

```

1  ///|
2  test {
3      let _map1 : @sorted_map.SortedMap[Int, String] = @sorted_map.new()
4      let _map2 = @sorted_map.from_array([(1, "one"), (2, "two"), (3, "three")])
5
6  }

```

Container Operations

Add a key-value pair to the SortedMap in place.

```

1  ///|
2  test {
3      let map = @sorted_map.from_array([(1, "one"), (2, "two")])
4      map.set(3, "three")
5      assert_eq(map.size(), 3)
6  }

```

You can also use the convenient subscript syntax to add or update values:

```

1  ///|
2  test {
3      let map = @sorted_map.new()
4      map[1] = "one"
5      map[2] = "two"
6      assert_eq(map.size(), 2)
7  }

```

Remove a key-value pair from the SortedMap in place.

```

1  ///|
2  test {
3      let map = @sorted_map.from_array([(1, "one"), (2, "two"), (3, "three")])
4      map.remove(2)
5      assert_eq(map.size(), 2)
6      assert_eq(map.contains(2), false)
7  }

```

Get a value by its key. The return type is Option[V].

```

1  ///|
2  test {
3      let map = @sorted_map.from_array([(1, "one"), (2, "two"), (3, "three")])
4      assert_eq(map.get(2), Some("two"))
5      assert_eq(map.get(4), None)
6  }

```

Safe access with error handling:

```

1  ///|
2  test {
3    let map = @sorted_map.from_array([(1, "one"), (2, "two")])
4    let key = 3
5    inspect(map.get(key), content="None")
6  }

```

Check if a key exists in the map.

```

1  ///|
2  test {
3    let map = @sorted_map.from_array([(1, "one"), (2, "two"), (3, "three")])
4    assert_eq(map.contains(2), true)
5    assert_eq(map.contains(4), false)
6  }

```

Iterate over all key-value pairs in the map in sorted key order.

```

1  ///|
2  test {
3    let map = @sorted_map.from_array([(3, "three"), (1, "one"), (2, "two")])
4    let keys = []
5    let values = []
6    map.each((k, v) => {
7      keys.push(k)
8      values.push(v)
9    })
10   assert_eq(keys, [1, 2, 3])
11   assert_eq(values, ["one", "two", "three"])
12 }

```

Iterate with index:

```

1  ///|
2  test {
3    let map = @sorted_map.from_array([(3, "three"), (1, "one"), (2, "two")])
4    let result = []
5    map.eachi((i, k, v) => result.push((i, k, v)))
6    assert_eq(result, [(0, 1, "one"), (1, 2, "two"), (2, 3, "three")])
7  }

```

Get the size of the map.

```

1  ///|
2  test {
3    let map = @sorted_map.from_array([(1, "one"), (2, "two"), (3, "three")])
4    assert_eq(map.size(), 3)
5  }

```

Check if the map is empty.

```

1  ///|
2  test {
3      let map : @sorted_map.SortedMap[Int, String] = @sorted_map.new()
4      assert_eq(map.is_empty(), true)
5  }

```

Clear the map.

```

1  ///|
2  test {
3      let map = @sorted_map.from_array([(1, "one"), (2, "two"), (3, "three")])
4      map.clear()
5      assert_eq(map.is_empty(), true)
6  }

```

Data Extraction

Get all keys or values from the map.

```

1  ///|
2  test {
3      let map = @sorted_map.from_array([(3, "three"), (1, "one"), (2, "two")])
4      assert_eq(map.keys_as_iter().collect(), [1, 2, 3])
5      assert_eq(map.values_as_iter().collect(), ["one", "two", "three"])
6  }

```

Convert the map to an array of key-value pairs.

```

1  ///|
2  test {
3      let map = @sorted_map.from_array([(3, "three"), (1, "one"), (2, "two")])
4      assert_eq(map.to_array(), [(1, "one"), (2, "two"), (3, "three")])
5  }

```

Range Operations

Get a subset of the map within a specified range of keys. The range is inclusive for both bounds [low, high].

```

1  ///|
2  test {
3      let map = @sorted_map.from_array([
4          (1, "one"),
5          (2, "two"),
6          (3, "three"),
7          (4, "four"),
8          (5, "five"),
9      ])
10     let range_items = []
11     map.range(2, 4).each((k, v) => range_items.push((k, v)))
12     assert_eq(range_items, [(2, "two"), (3, "three"), (4, "four")])
13 }

```

Edge cases for range operations:

- If low > high, returns an empty result
- If low or high are outside the map bounds, returns only pairs within valid bound
- The returned iterator preserves the sorted order of keys

```

1  ///|
2  /// Example with out-of-bounds range
3  test {
4    let map = @sorted_map.from_array([(1, "one"), (2, "two"), (3, "three")])
5    let range_items = []
6    map.range(0, 10).each((k, v) => range_items.push((k, v)))
7    assert_eq(range_items, [(1, "one"), (2, "two"), (3, "three")])
8
9    // Example with invalid range
10   let empty_range : Array[(Int, String)] = []
11   map.range(10, 5).each((k, v) => empty_range.push((k, v)))
12   assert_eq(empty_range, [])
13 }

```

Iterators

The SortedMap supports several iterator patterns. Create a map from an iterator:

```

1  ///|
2  test {
3    let pairs = [(1, "one"), (2, "two"), (3, "three")].iter()
4    let map = @sorted_map.from_iter(pairs)
5    assert_eq(map.size(), 3)
6  }

```

Use the iter method to get an iterator over key-value pairs:

```

1  ///|
2  test {
3    let map = @sorted_map.from_array([(3, "three"), (1, "one"), (2, "two")])
4    let pairs = map.iter().to_array()
5    assert_eq(pairs, [(1, "one"), (2, "two"), (3, "three")])
6  }

```

Use the iter2 method for a more convenient key-value iteration:

```

1  ///|
2  test {
3    let map = @sorted_map.from_array([(3, "three"), (1, "one"), (2, "two")])
4    let transformed = []
5    map.iter2().each((k, v) => transformed.push(k.to_string() + ": " + v))
6    assert_eq(transformed, ["1: one", "2: two", "3: three"])
7  }

```

Equality

Maps with the same key-value pairs are considered equal, regardless of the order in which elements were added.

```

1  ///|
2  test {
3      let map1 = @sorted_map.from_array([(1, "one"), (2, "two")])
4      let map2 = @sorted_map.from_array([(2, "two"), (1, "one")])
5      assert_eq(map1 == map2, true)
6  }

```

Error Handling Best Practices

When working with keys that might not exist, prefer using pattern matching for safety:

```

1  ///|
2  fn get_score(scores : @sorted_map.SortedMap[Int, Int], student_id : Int)
3      match scores.get(student_id) {
4      Some(score) => score
5      None =>
6          // println(
7          //     "Student ID " +
8          //     student_id.to_string() +
9          //     " does not exist, returning default score",
10         // )
11         0 // Default score
12     }
13 }
14
15 ///|
16 test "safe_key_access" {
17     // Create a mapping storing student IDs and their scores
18     let scores = @sorted_map.from_array([(1001, 85), (1002, 92), (1003, 78)
19
20     // Access an existing key
21     assert_eq(get_score(scores, 1001), 85)
22
23     // Access a non-existent key, returning the default value
24     assert_eq(get_score(scores, 9999), 0)
25 }

```

Implementation Notes

The SortedMap is implemented as an AVL tree, a self-balancing binary search tree. After insertions and deletions, the tree automatically rebalances to maintain $O(\log n)$ search, insertion, and deletion times.

Key properties of the AVL tree implementation:

- Each node stores a balance factor (height difference between left and right subtrees)
- The balance factor is maintained between -1 and 1 for all nodes
- Rebalancing is done through tree rotations (single and double rotations)

Comparison with Other Collections

- **@hashmap.T**: Provides $O(1)$ average case lookups but doesn't maintain order; use when order doesn't matter
- **@indexmap.T**: Maintains insertion order but not sorted order; use when insertion order matters
- **@sorted_map.SortedMap**: Maintains keys in sorted order; use when you need keys to be sorted

Choose SortedMap when you need:

- Key-value pairs sorted by key
- Efficient range queries
- Ordered traversal guarantees

MoonBit QuickCheck Package

MoonBit QuickCheck package provides property-based testing capabilities by generating random test inputs.

Basic Usage

Generate random values of any type that implements the Arbitrary trait:

```

1  ///|
2  test "basic generation" {
3      let b : Bool = @quickcheck.gen()
4      inspect(b, content="true")
5      let x : Int = @quickcheck.gen()
6      inspect(x, content="0")
7
8      // Generate with size parameter
9      let sized : Array[Int] = @quickcheck.gen(size=5)
10     inspect(sized.length() <= 5, content="true")
11 }

```

Multiple Samples

Generate multiple test cases using the samples function:

```

1  ///|
2  test "multiple samples" {
3      let ints : Array[Int] = @quickcheck.samples(5)
4      inspect(ints, content="[0, 0, 0, -1, -1]")
5      let strings : Array[String] = @quickcheck.samples(12)
6      inspect(
7          strings[5:10],
8          content=(
9              #|["E\b\u{0f} ", " ", "K\u{1f}[" , " !@", "xvLxb"]
10             ),
11      )
12 }

```

Built-in Types

QuickCheck provides Arbitrary implementations for all basic MoonBit types:

```
1  ///|
2  test "builtin types" {
3    // Basic types
4    let v : (Bool, Char, Byte) = @quickcheck.gen()
5    inspect(v, content="(true, '#', b'\\x12')")
6    // Numeric types
7    let v : (Int, Int64, UInt, UInt64, Float, Double, BigInt) = @quickcheck.gen()
8    inspect(v, content="(0, 0, 0, 0, 0.1430625319480896, 0.330984466952546, 0)")
9    // Collections
10
11    let v : (String, Bytes, Iter[Int]) = @quickcheck.gen()
12    inspect(
13      v,
14      content=(
15        #|(" ", b"", [])
16      ),
17    )
18  }
```

Custom Types

Implement Arbitrary trait for custom types:

```
1  ///|
2  struct Point {
3    x : Int
4    y : Int
5  } derive(Show)
6
7  ///|
8  impl Arbitrary for Point with arbitrary(size, r0) {
9    let r1 = r0.split()
10    let y = @quickcheck.Arbitrary::arbitrary(size, r1)
11    { x: @quickcheck.Arbitrary::arbitrary(size, r0), y }
12  }
13
14  ///|
15  test "custom type generation" {
16    let point : Point = @quickcheck.gen()
17    inspect(point, content="{x: 0, y: 0}")
18    let points : Array[Point] = @quickcheck.samples(10)
19    inspect(
20      points[6:],
21      content="[ {x: 0, y: 1}, {x: -1, y: -5}, {x: -6, y: -6}, {x: -1, y: 7} ]"
22    )
23  }
```

The package is useful for writing property tests that verify code behavior across a wide range of randomly generated inputs.

uint

This package provides functionalities for handling 32-bit unsigned integers in MoonBit. To this end, it includes methods for converting between UInt and other number formats, as well as utilities for byte representation.

Basic Properties

uint provides constants for UInt's value range and default value:

```
1  ///|
2  test "uint basics" {
3    // Default value is 0
4    inspect(@uint.default(), content="0")
5
6    // Maximum and minimum values
7    inspect(@uint.max_value, content="4294967295")
8    inspect(@uint.min_value, content="0")
9  }
```

Byte Representation

UInt can be converted to bytes in both big-endian and little-endian formats:

```
1  ///|
2  test "uint byte conversion" {
3    let num = 258U // 0x00000102 in hex
4
5    // Big-endian bytes (most significant byte first)
6    let be_bytes = num.to_be_bytes()
7    inspect(
8      be_bytes,
9      content=(
10       #|b"\x00\x00\x01\x02"
11     ),
12   )
13
14   // Little-endian bytes (least significant byte first)
15   let le_bytes = num.to_le_bytes()
16   inspect(
17     le_bytes,
18     content=(
19       #|b"\x02\x01\x00\x00"
20     ),
21   )
22 }
```

Converting to Other Number Types

UInt can be converted to Int64 when you need to work with signed 64-bit integers
:

```
1  ///|
2  test "uint type conversion" {
3    let num = 42U
4    inspect(num.to_int64(), content="42")
5    let large_num = 4294967295U // max value
6    inspect(large_num.to_int64(), content="4294967295")
7  }
```

These conversion functions are also available as methods:

```
1  ///|
2  test "uint methods" {
3    let num = 1000U
4
5    // Using method syntax
6    inspect(num.to_int64(), content="1000")
7    inspect(
8      num.to_be_bytes(),
9      content=(
10       #|b"\x00\x00\x03\xe8"
11     ),
12   )
13   inspect(
14     num.to_le_bytes(),
15     content=(
16       #|b"\xe8\x03\x00\x00"
17     ),
18   )
19 }
```

json

The json package provides comprehensive JSON handling capabilities, including parsing, stringifying, and type-safe conversion between JSON and other MoonBit data types.

Basic JSON Operations

Parsing and Validating JSON

```

1  ///|
2  test "parse and validate jsons" {
3    // Check if a string is valid JSON
4    assert_true(@json.valid("{\"key\": 42}"))
5    assert_true(@json.valid("[1, 2, 3]"))
6    assert_true(@json.valid("null"))
7    assert_true(@json.valid("false"))
8
9    // Parse JSON string into Json value
10   let json = @json.parse("{\"key\": 42}") catch {
11     (_ : @json.ParseError) => panic()
12     // _ => panic() // redundant, the type checker won't refine further
13   }
14
15   // Pretty print with indentation
16   inspect(
17     json.stringify(indent=2),
18     content={
19       let output =
20         #|{
21         #|  "key": 42
22         #|}
23       output
24     },
25   )
26 }

```

Object Navigation

```

1  ///|
2  test "json object navigation" {
3      let json = @json.parse(
4          "{ \"string\": \"hello\", \"number\": 42, \"array\": [1, 2, 3] }",
5      )
6
7      // Access string
8      let string_opt = json.value("string").unwrap().as_string()
9      inspect(
10         string_opt,
11         content=(
12             #|Some("hello")
13         ),
14     )
15
16     // Access number
17     let number_opt = json.value("number").unwrap().as_number()
18     inspect(number_opt, content="Some(42)")
19
20     // Access array
21     let array_opt = json.value("array").unwrap().as_array()
22     inspect(array_opt, content="Some([Number(1), Number(2), Number(3)])")
23
24     // Handle missing keys gracefully
25     inspect(json.value("missing"), content="None")
26 }

```

Array Navigation

```

1  ///|
2  test "json array navigation" {
3      let array = @json.parse("[1, 2, 3, 4, 5]")
4
5      // Access by index
6      let first = array.item(0)
7      inspect(first, content="Some(Number(1))")
8
9      // Access out of bounds
10     let missing = array.item(10)
11     inspect(missing, content="None")
12
13     // Iterate through array
14     let values = array.as_array().unwrap()
15     inspect(
16         values.iter(),
17         content="[Number(1), Number(2), Number(3), Number(4), Number(5)]",
18     )
19 }

```

Type-Safe JSON Conversion

From JSON to Native Types

```

1  ///|
2  test "json decode" {
3    // Decode basic types
4    let json_number = (42 : Json)
5    let number : Int = @json.from_json(json_number)
6    inspect(number, content="42")
7
8    // Decode arrays
9    let json_array = ([1, 2, 3] : Json)
10   let array : Array[Int] = @json.from_json(json_array)
11   inspect(array, content="[1, 2, 3]")
12
13   // Decode maps
14   let json_map = ({ "a": 1, "b": 2 } : Json)
15   let map : Map[String, Int] = @json.from_json(json_map)
16   inspect(
17     map,
18     content=(
19       #|{"a": 1, "b": 2}
20     ),
21   )
22 }

```

Error Handling with JSON Path

```

1  ///|
2  test "json path" {
3    // Handle decode errors
4    try {
5      let _arr : Array[Int] = @json.from_json([42, "not a number", 49] :
6      panic()
7    } catch {
8      @json.JsonDecodeError((path, msg)) => {
9        inspect(path, content="$[1]")
10       inspect(msg, content="Int::from_json: expected number")
11     }
12   }
13 }

```

JSON-based Snapshot Testing

@json.inspect() can be used as an alternative to inspect() when a value's To Json implementation is considered a better debugging representation than its Show implementation. This is particularly true for deeply-nested data structures.

```

1  ///|
2  test "json inspection" {
3    let null = null
4
5    // Simple json values
6    let json_value : Json = { "key": "value", "numbers": [1, 2, 3] }
7    @json.inspect(json_value, content={ "key": "value", "numbers": [1, 2,
8
9    // Null and boolean values
10   let json_special = { "null": null, "bool": true }
11   @json.inspect(json_special, content={ "null": null, "bool": true })
12 }

```

Env Package Documentation

This package provides utilities for interacting with the runtime environment, including access to command line arguments, current time, and working directory information.

Command Line Arguments

Access command line arguments passed to your program:

```

1  ///|
2  test "command line arguments" {
3    let arguments = @env.args()
4
5    // The arguments array contains program arguments
6    // In a test environment, this will typically be empty or contain test
7    inspect(arguments.length() >= 0, content="true")
8
9    // Example of how you might process arguments in a real program:
10   fn process_args(args : Array[String]) -> String {
11     if args.length() == 0 {
12       "No arguments provided"
13     } else {
14       "First argument: " + args[0]
15     }
16   }
17
18   let result = process_args(arguments)
19   inspect(result.length() > 0, content="true")
20 }

```

Current Time

Get the current time in milliseconds since Unix epoch:


```

1  ///|
2  test "current time" {
3    let timestamp = @env.now()
4
5    // Timestamp should be a reasonable value (after year 2020)
6    let year_2020_ms = 1577836800000UL // Jan 1, 2020 in milliseconds
7    inspect(timestamp > year_2020_ms, content="true")
8
9    // Demonstrate time-based operations
10   fn format_timestamp(ts : UInt64) -> String {
11     "Timestamp: " + ts.to_string()
12   }
13
14   let formatted = format_timestamp(timestamp)
15   inspect(formatted.length() > 10, content="true") // Should contain time
16 }

```

Working Directory

Get the current working directory:

```

1  ///|
2  test "working directory" {
3    let cwd = @env.current_dir()
4    match cwd {
5      Some(path) => {
6        // We have a current directory
7        inspect(path.length() > 0, content="true")
8        inspect(path.length() > 1, content="true") // Should be a meaningful path
9      }
10     None =>
11       // Current directory unavailable (some platforms/environments)
12       inspect(true, content="true") // This is also valid
13   }
14 }

```

Practical Usage Examples

Command Line Tool Pattern

```

1  ///|
2  test "command line tool pattern" {
3      fn parse_command(args : Array[String]) -> Result[String, String] {
4          if args.length() < 2 {
5              Err("Usage: program <command> [args...]")
6          } else {
7              match args[1] {
8                  "help" => Ok("Showing help information")
9                  "version" => Ok("Version 1.0.0")
10                 "status" => Ok("System is running")
11                 cmd => Err("Unknown command: " + cmd)
12             }
13         }
14     }
15
16     // Test with mock arguments
17     let test_args = ["program", "help"]
18     let result = parse_command(test_args)
19     inspect(result, content="Ok(\"Showing help information\")")
20     let invalid_result = parse_command(["program", "invalid"])
21     match invalid_result {
22         Ok(_) => inspect(false, content="true")
23         Err(msg) => inspect(msg.length() > 10, content="true") // Should have
24     }
25 }

```

Configuration Loading

```

1  ///|
2  test "configuration loading" {
3      fn load_config_path() -> String {
4          match @env.current_dir() {
5              Some(cwd) => cwd + "/config.json"
6              None => "./config.json" // Fallback
7          }
8      }
9
10     let config_path = load_config_path()
11     inspect(config_path.length() > 10, content="true") // Should have path
12 }

```

Logging with Timestamps

```

1  ///|
2  test "logging with timestamps" {
3      fn log_message(level : String, message : String) -> String {
4          let timestamp = @env.now()
5          "[" + timestamp.to_string() + "] " + level + ": " + message
6      }
7
8      let log_entry = log_message("INFO", "Application started")
9      inspect(log_entry.length() > 20, content="true") // Should have timestamp
10     inspect(log_entry.length() > 10, content="true") // Should have substantial
11 }

```

File Path Operations

```
1  ///|
2  test "file path operations" {
3    fn resolve_relative_path(relative : String) -> String {
4      match @env.current_dir() {
5        Some(base) => base + "/" + relative
6        None => relative
7      }
8    }
9
10   let resolved = resolve_relative_path("data/input.txt")
11   inspect(resolved.length() > 10, content="true") // Should have resolved
12 }
```

Platform Differences

The env package behaves differently across platforms:

JavaScript Environment

- args() returns arguments from the JavaScript environment
- @env.now() uses Date.now()
- @env.current_dir() may return None in browser environments

WebAssembly Environment

- args() behavior depends on the WASM host
- @env.now() provides millisecond precision timing
- @env.current_dir() availability depends on host capabilities

Native Environment

- args() returns actual command line arguments
- @env.now() provides system time
- @env.current_dir() uses system calls to get working directory

Error Handling

Handle cases where environment information is unavailable:

```

1  ///|
2  test "error handling" {
3    fn safe_get_cwd() -> String {
4      match @env.current_dir() {
5        Some(path) => path
6        None =>
7          // Fallback when current directory is unavailable
8          "."
9      }
10   }
11
12   let safe_cwd = safe_get_cwd()
13   inspect(safe_cwd.length() > 0, content="true")
14   fn validate_args(
15     args : Array[String],
16     min_count : Int,
17   ) -> Result[Unit, String] {
18     if args.length() < min_count {
19       Err("Insufficient arguments: expected at least " + min_count.to_st
20     } else {
21       Ok(())
22     }
23   }
24
25   let validation = validate_args(["prog"], 2)
26   match validation {
27     Ok(_) => inspect(false, content="true")
28     Err(msg) => inspect(msg.length() > 10, content="true") // Should hav
29   }
30 }

```

Best Practices

1. Handle Missing Environment Data Gracefully

```

1  ///|
2  test "graceful handling" {
3    fn get_work_dir() -> String {
4      match @env.current_dir() {
5        Some(dir) => dir
6        None => "~" // Fallback to home directory symbol
7      }
8    }
9
10   let work_dir = get_work_dir()
11   inspect(work_dir.length() > 0, content="true")
12 }

```

2. Validate Command Line Arguments

```

1  ///|
2  test "argument validation" {
3    fn validate_and_parse_args(
4      args : Array[String],
5    ) -> Result[(String, Array[String]), String] {
6      if args.length() == 0 {
7        Err("No program name available")
8      } else if args.length() == 1 {
9        Ok((args[0], [])) // Program name only, no arguments
10     } else {
11       let program = args[0]
12       let arguments = Array::new()
13       for i in 1..<args.length() {
14         arguments.push(args[i])
15       }
16       Ok((program, arguments))
17     }
18   }
19
20   let test_result = validate_and_parse_args(["myprogram", "arg1", "arg2"])
21   match test_result {
22     Ok((prog, args)) => {
23       inspect(prog, content="myprogram")
24       inspect(args.length(), content="2")
25     }
26     Err(_) => inspect(false, content="true")
27   }
28 }

```

3. Use Timestamps for Unique Identifiers

```

1  ///|
2  test "unique identifiers" {
3    fn generate_unique_id(prefix : String) -> String {
4      prefix + "_" + @env.now().to_string()
5    }
6
7    let id1 = generate_unique_id("task")
8    let id2 = generate_unique_id("task")
9    inspect(id1.length() > 10, content="true") // Should have task prefix
10   inspect(id2.length() > 10, content="true") // Should have task prefix
11   // IDs should be different (though they might be the same in fast test
12 }

```

Common Use Cases

- **Command Line Tools:** Parse arguments and provide help/usage information
- **Configuration Management:** Load config files relative to current directory
- **Logging Systems:** Add timestamps to log entries
- **File Processing:** Resolve relative file paths
- **Debugging:** Include environment information in error reports
- **Build Tools:** Determine working directory for relative path operations

Performance Considerations

- `args()` is typically called once at program startup
- `@env.now()` is lightweight but avoid calling in tight loops if high precision isn't needed
- `@env.current_dir()` may involve system calls, so cache the result if used frequently
- Environment functions are generally fast but platform-dependent

The `env` package provides essential runtime environment access for building robust MoonBit applications that interact with their execution environment.