Functions

Functions are first-class values in Epos, supporting higher-order functions, generics, and default parameters.

Basic Function Syntax

Functions use the `fn` keyword (short for function) with implicit returns (last expression is returned):

```
fn say-hello(name: string)
    print("Hello, #{name}!")
end

fn add(a: int, b: int): int
    a + b
end

fn greet(name: string): string
    "Hello, #{name}!"
end

fn multiply(x: int, y: int): int
    x * y
end
```

Function Types

Functions are values and can be passed around:

```
# Function type syntax: fn(param_types) -> return_type
fn higher-order(f: fn(int, int) -> int, x: int, y: int): int
    f(x, y)
end

# Use it
result: int = higher-order(add, 5, 3)  # result is 8
```

Lambda Functions

Create anonymous functions using lambda syntax:

```
# Lambda with explicit type
square := fn(x: int): int => x * x

# Use lambdas with higher-order functions
numbers := {1, 2, 3, 4, 5}
squared := numbers.map(fn(x: int): int => x * x)
```

Default Parameters

Functions can have default parameter values:

```
fn greet-with-title(name: string, title: string = "Mr."): string
    "Hello, #{title} #{name}!"
end

# Call with default
greeting1 := greet-with-title("Smith")        # "Hello, Mr. Smith!"

# Call with explicit value
greeting2 := greet-with-title("Smith", "Dr.") # "Hello, Dr. Smith!"
```

Generic Functions

Write functions that work with any type:

```
fn identity(value: t): t
    value
end

# Generic function with multiple type parameters
fn make-pair(first: a, second: b): Pair(a, b)
    @{
        first => first,
        second => second
    }
end

# Usage
name-age := make-pair("Alice", 30)
coords := make-pair(10, 20)
```

Higher-Order Functions

Functions that take or return other functions:

```
# Function that returns a function
fn make-adder(n: int): fn(int) => int
    fn(x: int) => x + n
end

add-five := make-adder(5)
result := add-five(10)    # result is 15

# Function composition
fn compose(f: fn(b) -> c, g: fn(a) -> b): fn(a) -> c
    fn(x: a) => f(g(x))
end
```

Recursive Functions

Functions can call themselves:

```
fn factorial(n: int): int
    match n <= 1 then
        true => 1
        false => n * factorial(n - 1)
    end
end

fn fibonacci(n: int): int
    match n then
        0, 1 => n
        _ => fibonacci(n - 1) + fibonacci(n - 2)
    end
end
```

Built-in List Functions

Epos provides several built-in functions for working with lists:

```
numbers: list(int) = {1, 2, 3, 4, 5}
```

```
# Map: transform each element
doubled := numbers.map(fn(x: int): int => x * 2)

# Filter: select elements that match a predicate
evens := numbers.filter(fn(x: int): bool => x % 2 == 0)

# Each: iterate over list elements
numbers.each(fn(x: int) => print(x))

# Length: get number of elements
count := numbers.len()

# Element access
first := numbers.elem(0)
```

Type Aliases for Function Types

Make complex function types more readable:

```
type Predicate(t) = fn(t) -> bool
type Transform(a, b) = fn(a) -> b
type BinaryOp(t) = fn(t, t) -> t

fn filter-with-predicate(items: list(t), pred: Predicate(t)): list(t)
    filter(items, pred)
end
```

Next, learn about pattern matching in Epos.