Advanced Features

This guide covers advanced Epos features including generics, higher-order programming, and working with complex data flows.

Generic Programming

Generic Functions with Multiple Parameters

Create functions that work with multiple generic types:

```
fn zip(first: list(a), second: list(b)): list(Pair(a, b))
  # Implementation would recursively build pairs
  {}  # Placeholder
end

fn fold(items: list(t), initial: a, combiner: fn(a, t) => a): a
  match len(items) then
    0 => initial
    _ => fold(tail(items), combiner(initial, head(items)), combiner)
  end
end
```

Generic Type Constraints

While Epos doesn't have explicit constraints, patterns emerge for type-safe generic code:

```
# Generic container with operations
record Container(t)
  items: list(t)
  size: int
end

fn add-to-container(container: Container(t), item: t): Container(t)
  @{
    items => append(container.items, item),
    size => container.size + 1
  }
end
```

Advanced Pattern Matching

Matching with Complex Conditions

```
record Result(t, e)
  value?: t
  error?: e
  is-success: bool
end

fn handle-result(result: Result(string, string)): string
  match result.is-success then
    true => match result.value then
      some-value => "Success: #{some-value}"
      _ => "Success but no value"
    end
    false => match result.error then
      some-error => "Error: #{some-error}"
      _ => "Unknown error"
    end
```

```
  end
end
```

Pattern Matching with Deconstruction

```
# Match and extract from complex structures
fn process-request(request: HttpRequest): HttpResponse
  match request.method then
    "GET" => handle-get(request.path, request.params)
    "POST" => handle-post(request.path, request.body)
    "PUT" => handle-put(request.path, request.body)
    _ => @{
      status => 405,
      body => "Method not allowed"
    }
  end
end
```

Higher-Order Programming Patterns

Function Composition and Chaining

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

# Pipe operator simulation
fn pipe(value: a, transform: fn(a) => b): b
  transform(value)
end

# Chaining transformations
result := numbers
  .pipe(fn(nums: list(int)) => filter(nums, fn(n: int) => n > 0))
  .pipe(fn(nums: list(int)) => map(nums, fn(n: int) => n * 2))
  .pipe(fn(nums: list(int)) => fold(nums, 0, fn(acc: int, n: int) => acc + n))
```

Currying and Partial Application

```
# Curried functions
fn curried-add(a: int): fn(int) => int
  fn(b: int) => a + b
end

add-five := curried-add(5)
result := add-five(10)  # 15

# Partial application pattern
fn make-validator(min-length: int): fn(string) => bool
  fn(input: string) => len(input) >= min-length
end

validate-password := make-validator(8)
is-valid := validate-password("mypassword")
```

Working with Lists - Advanced Operations

Custom List Operations

```
# Take first n elements
fn take(items: list(t), count: int): list(t)
  match count <= 0 || len(items) == 0 then
    true => {}
    false => {head(items), ..take(tail(items), count - 1)}
  end
end

# Drop first n elements
fn drop(items: list(t), count: int): list(t)
  match count <= 0 then
    true => items
    false => match len(items) then
      0 => {}
      _ => drop(tail(items), count - 1)
    end
  end
end

# Group consecutive elements
fn group-by(items: list(t), key-fn: fn(t) => k): list(list(t))
  # Implementation would group items by key function result
  {}  # Placeholder
end
```

List Processing Pipelines

```
# Complex data processing pipeline
fn process-data(raw-data: list(string)): list(ProcessedRecord)
  raw-data
    .filter(fn(line: string) => len(line) > 0)     # Remove empty lines
    .map(fn(line: string) => parse-csv-line(line))  # Parse each line
    .filter(fn(record: RawRecord) => validate-record(record))  # Validate
    .map(fn(record: RawRecord) => transform-record(record))    # Transform
end
```

Error Handling Patterns

Option Type Pattern

```
record Option(t)
  value?: t
  has-value: bool
end

fn some(value: t): Option(t)
  @{
    value => value,
    has-value => true
  }
end

fn none(): Option(t)
  @{
    has-value => false
```

```
  }
end

fn map-option(opt: Option(a), transform: fn(a) => b): Option(b)
  match opt.has-value then
    true => some(transform(opt.value))
    false => none()
  end
end
```

Result Type Pattern

```
record Result(t, e)
  value?: t
  error?: e
  is-success: bool
end

fn ok(value: t): Result(t, e)
  @{
    value => value,
    is-success => true
  }
end

fn err(error: e): Result(t, e)
  @{
    error => error,
    is-success => false
  }
end
```

Module-like Organization

Namespace Patterns

```
# Group related functions using record-like syntax
math-utils := @{
  add => fn(a: int, b: int): int => a + b,
  multiply => fn(a: int, b: int): int => a * b,
  power => fn(base: int, exp: int): int =>
    match exp then
      0 => 1
      _ => base * math-utils.power(base, exp - 1)
    end
}

# Usage
result := math-utils.add(5, 3)
squared := math-utils.power(4, 2)
```

Performance Patterns

Tail Recursion

```
# Tail-recursive factorial
fn factorial-tail(n: int, acc: int = 1): int
  match n <= 1 then
    true => acc
```

```
    false => factorial-tail(n - 1, n * acc)
  end
end

# Tail-recursive list processing
fn reverse-tail(items: list(t), acc: list(t) = {}): list(t)
  match len(items) then
    0 => acc
    _ => reverse-tail(tail(items), {head(items), ..acc})
  end
end
```

Lazy Evaluation Simulation

```
# Lazy computation using functions
record Lazy(t)
  compute: fn() => t
  computed?: t
  is-computed: bool
end

fn lazy(computation: fn() => t): Lazy(t)
  @{
    compute => computation,
    is-computed => false
  }
end

fn force(lazy-val: Lazy(t)): t
  match lazy-val.is-computed then
    true => lazy-val.computed
    false => lazy-val.compute()
  end
end
```

These advanced features enable sophisticated functional programming patterns while maintaining Epos's clean syntax and type safety.