Setsuna Programming Language

A Modern Functional Programming Language

Language Reference Manual

December 3, 2025

Abstract

Setsuna is a modern functional programming language that combines ML-family language features with C-style syntax. It features Hindley-Milner type inference, powerful pattern matching, algebraic data types, first-class functions, and an immutability-first design philosophy. This document provides a comprehensive overview of the language's syntax, semantics, and standard library.

Contents

1 Introduction

Setsuna is a functional programming language designed to make functional programming natural and intuitive while maintaining a familiar syntax. The language draws inspiration from ML-family languages (such as OCaml and Haskell) while adopting a C-style syntax that is accessible to programmers from imperative backgrounds.

1.1 Design Philosophy

The core design principles of Setsuna are:

1. **Functional First**: Make functional programming patterns natural and idiomatic
2. **Pattern Matching**: Provide powerful pattern matching with destructuring capabilities
3. **Type Safety**: Automatic type inference with compile-time safety guarantees
4. **Clean Syntax**: Expressive yet familiar syntax inspired by C-family languages
5. **Immutability**: Favor immutable data structures and pure functions
6. **Interactive Development**: Built-in REPL support for exploratory programming

1.2 File Extension

Setsuna source files use the `.stsn` file extension.

2 Lexical Structure

2.1 Comments

Setsuna supports single-line comments using the `//` prefix:

```
// This is a single-line comment
let x = 42  // Inline comment
```

2.2 Keywords

The following identifiers are reserved keywords in Setsuna:

`let`	`fn`	`if`	`else`	`match`	`type`
`module`	`import`	`true`	`false`		

2.3 Operators

2.3.1 Arithmetic Operators

Operator	Description
`+`	Addition
`-`	Subtraction (binary) / Negation (unary)
`*`	Multiplication
`/`	Division
`%`	Modulo

2.3.2 Comparison Operators

Operator	Description
`==`	Equal
`!=`	Not equal
`<`	Less than
`>`	Greater than
`<=`	Less than or equal
`>=`	Greater than or equal

2.3.3 Logical Operators

Operator	Description
`&&`	Logical AND (short-circuit)
`||`	Logical OR (short-circuit)
`!`	Logical NOT

2.3.4 Operator Precedence

From highest to lowest precedence:

1. `.` (field/module access)
2. `()` (function call)
3. `!`, `-` (unary operators)
4. `*`, `/`, `%`
5. `+`, `-`
6. `<`, `>`, `<=`, `>=`
7. `==`, `!=`
8. `&&`
9. `||`
10. `=>` (right associative)

3 Data Types

Setsuna features automatic type inference using the Hindley-Milner algorithm, so explicit type annotations are rarely needed.

3.1 Primitive Types

3.1.1 Integers

64-bit signed integers:

```
let a = 42
let b = -17
let large = 9223372036854775807
```

3.1.2 Floating-Point Numbers

64-bit floating-point (IEEE 754 double precision):

```
let pi = 3.14159
let e = 2.71828
let negative = -0.5
```

3.1.3 Booleans

Boolean values with literals `true` and `false`:

```
let yes = true
let no = false
let result = 5 > 3  // true
```

3.1.4 Strings

UTF-8 encoded strings with escape sequences:

```
let greeting = "Hello, World!"
let multiline = "Line 1\nLine 2"
let escaped = "She said \"Hello\""
```

3.1.5 Unit

The unit type represents the absence of a meaningful value:

```
let nothing = ()
```

3.2 Compound Types

3.2.1 Lists

Homogeneous, ordered collections:

```
let numbers = [1, 2, 3, 4, 5]
let empty = []
let words = ["apple", "banana", "cherry"]
let nested = [[1, 2], [3, 4], [5, 6]]
```

3.2.2 Tuples

Fixed-size collections that can hold heterogeneous types:

```
let point = (10, 20)
let person = ("Alice", 30, true)
let single = (42,)  // Single-element tuple
let access = point.0  // Accessing first element
```

3.2.3 Records

Collections with named fields:

```
let person = {
    name: "Alice",
    age: 30,
    city: "Tokyo"
}

// Field access
let name = person.name
let age = person.age
```

4 Variables and Bindings

Variables in Setsuna are immutable by default and are introduced using the `let` keyword.

4.1 Simple Bindings

```
let x = 42
let name = "Alice"
let is_valid = true
```

4.2 Pattern Bindings

The `let` keyword supports destructuring patterns:

```
// Tuple destructuring
let (x, y) = (10, 20)

// List destructuring
let [first, ...rest] = [1, 2, 3, 4, 5]

// Record destructuring
let { name: n, age: a } = { name: "Bob", age: 25 }
```

5 Functions

Functions are first-class values in Setsuna and can be passed as arguments, returned from other functions, and stored in data structures.

5.1 Function Definition

Functions are defined using the `fn` keyword. Two syntactic forms are available:

5.1.1 Block Syntax

```
fn add(a, b) {
    a + b
}

fn greet(name) {
    println("Hello, " + name)
```

```
}
```

5.1.2 Arrow Syntax

For single-expression functions:

```
fn square(x) => x * x
fn double(x) => x * 2
fn greet(name) => "Hello, " + name
```

5.2 Lambda Expressions

Anonymous functions (lambdas) use arrow syntax:

```
let double = (x) => x * 2
let add = (a, b) => a + b

// Multi-expression lambda with block
let quad = (x) => {
    let squared = x * x
    squared * squared
}
```

5.3 Higher-Order Functions

Functions can accept and return other functions:

```
fn apply_twice(f, x) {
    f(f(x))
}

let result = apply_twice((x) => x + 1, 10)  // 12

fn make_adder(n) {
    (x) => x + n
}

let add5 = make_adder(5)
add5(10)  // 15
```

5.4 Closures

Functions capture variables from their enclosing scope:

```
fn counter(start) {
    let count = start
    () => {
        count + 1
    }
}

fn make_multiplier(factor) {
    (x) => x * factor
}

let triple = make_multiplier(3)
triple(7)  // 21
```

5.5 Function Composition

```
fn compose(f, g) => (x) => f(g(x))
fn pipe(f, g) => (x) => g(f(x))

let double = (x) => x * 2
let inc = (x) => x + 1

let double_then_inc = pipe(double, inc)
double_then_inc(5)  // 11
```

5.6 Recursion

Setsuna supports recursive functions. The compiler optimizes tail-recursive calls:

```
// Simple recursion
fn factorial(n) {
    match n {
        0 => 1,
        _ => n * factorial(n - 1)
    }
}

// Tail-recursive version
fn factorial_tail(n) {
    fn helper(n, acc) {
        match n {
            0 => acc,
            _ => helper(n - 1, n * acc)
        }
    }
    helper(n, 1)
}
```

6 Control Flow

All control flow constructs in Setsuna are expressions that return values.

6.1 If Expressions

```
let result = if x > 5 { "big" } else { "small" }

// Chained conditionals
fn grade(score) {
    if score >= 90 { "A" }
    else if score >= 80 { "B" }
    else if score >= 70 { "C" }
    else if score >= 60 { "D" }
    else { "F" }
}
```

6.2 Match Expressions

Pattern matching is the primary mechanism for control flow:

```
match value {
    0 => "zero",
    1 => "one",
    n if n > 10 => "large",
    _ => "other"
}
```

6.3 Block Expressions

Blocks are expressions that return the value of their last expression:

```
let result = {
    let a = 10
    let b = 20
    a + b  // Block returns 30
}
```

7 Pattern Matching

Pattern matching is a powerful feature in Setsuna that allows for concise and expressive code.

7.1 Pattern Types

7.1.1 Literal Patterns

```
match x {
    0 => "zero",
    1 => "one",
    "hello" => "greeting",
    true => "yes",
    _ => "other"
}
```

7.1.2 Variable Patterns

```
match x {
    n => n + 1  // Binds x to n
}
```

7.1.3 Wildcard Pattern

```
match x {
    _ => "matches anything"
}
```

7.1.4 List Patterns

```
match list {
    [] => "empty",
    [x] => "single element",
    [x, y] => "two elements",
```

```
    [h, ...t] => "head and tail"
}
```

7.1.5 Tuple Patterns

```
match point {
    (0, 0) => "origin",
    (x, 0) => "on x-axis",
    (0, y) => "on y-axis",
    (x, y) => "general point"
}
```

7.1.6 Record Patterns

```
match person {
    { name: "Alice", age: a } => "Alice is " + str(a),
    { name: n, age: a } if a >= 18 => n + " is an adult",
    { name: n, age: _ } => n + " is a minor"
}
```

7.1.7 Constructor Patterns

```
match option {
    Some(x) => "has value: " + str(x),
    None => "empty"
}
```

7.2 Pattern Guards

Guards add conditional constraints to patterns:

```
fn classify(n) {
    match n {
        x if x < 0 => "negative",
        x if x == 0 => "zero",
        x if x < 10 => "small positive",
        _ => "large positive"
    }
}
```

7.3 Exhaustive Matching

The compiler ensures all patterns are covered:

```
type Result {
    Ok(value),
    Err(message)
}

fn handle(result) {
    match result {
        Ok(v) => v,
        Err(e) => error(e)
```

```
        // All cases covered
    }
}
```

8 Algebraic Data Types

Algebraic Data Types (ADTs) allow defining custom types with multiple constructors.

8.1 Type Definition

```
type Option {
    Some(value),
    None
}

type Result {
    Ok(value),
    Err(message)
}

type List {
    Nil,
    Cons(head, tail)
}

type Tree {
    Leaf(value),
    Node(left, right)
}
```

8.2 Using ADTs

```
// Creating values
let maybe_value = Some(42)
let nothing = None
let ok_result = Ok("success")
let err_result = Err("something went wrong")

// Pattern matching on ADTs
fn unwrap_or(opt, default) {
    match opt {
        Some(x) => x,
        None => default
    }
}

fn map_option(f, opt) {
    match opt {
        Some(x) => Some(f(x)),
        None => None
    }
}
```

8.3 Recursive ADTs

```
type Tree {
    Leaf(value),
    Node(left, right)
}

fn tree_sum(tree) {
    match tree {
        Leaf(x) => x,
        Node(left, right) => tree_sum(left) + tree_sum(right)
    }
}

fn tree_map(f, tree) {
    match tree {
        Leaf(x) => Leaf(f(x)),
        Node(left, right) => Node(
            tree_map(f, left),
            tree_map(f, right)
        )
    }
}

// Example usage
let my_tree = Node(
    Node(Leaf(1), Leaf(2)),
    Node(Leaf(3), Leaf(4))
)

tree_sum(my_tree)  // 10
```

9 Modules

Modules provide namespace organization and encapsulation.

9.1 Module Definition

```
module Math {
    fn square(x) => x * x
    fn cube(x) => x * x * x

    fn distance(x1, y1, x2, y2) {
        let dx = x2 - x1
        let dy = y2 - y1
        sqrt(dx * dx + dy * dy)
    }
}
```

9.2 Module Access

```
Math.square(5)      // 25
Math.cube(3)        // 27
Math.distance(0, 0, 3, 4)  // 5.0
```

9.3 Module with Types

```
module User {
    type Status {
        Active,
        Inactive,
        Banned(reason)
    }

    fn create(name, email) {
        {
            name: name,
            email: email,
            status: Active
        }
    }

    fn is_active(user) {
        match user.status {
            Active => true,
            _ => false
        }
    }
}

let user = User.create("Alice", "alice@example.com")
User.is_active(user)  // true
```

9.4 Imports

```
import Math
import User

// After import, can use directly
let x = square(5)
let u = create("Bob", "bob@example.com")
```

10 Standard Library

Setsuna includes a comprehensive standard library of functions.

10.1 List Functions

10.1.1 Basic Operations

Function	Description
`head(lst)`	Returns the first element
`tail(lst)`	Returns all elements except the first
`cons(x, lst)`	Prepends x to the list
`len(lst)`	Returns the length of the list
`empty(lst)`	Returns true if the list is empty
`reverse(lst)`	Reverses the list
`append(a, b)`	Concatenates two lists
`nth(lst, n)`	Returns the n-th element (0-indexed)
`range(start, end)`	Creates a list from start to end-1
`sort(lst)`	Sorts the list

10.1.2 Higher-Order Functions

Function	Description
`map(f, lst)`	Applies f to each element
`filter(pred, lst)`	Keeps elements where pred is true
`fold(f, init, lst)`	Left fold with accumulator
`fold_right(f, lst, init)`	Right fold with accumulator
`reduce(f, lst)`	Reduces without initial value
`flat_map(f, lst)`	Maps and flattens results
`any(pred, lst)`	True if any element satisfies pred
`all(pred, lst)`	True if all elements satisfy pred
`find(pred, lst)`	Finds first element satisfying pred

10.1.3 Slicing and Zipping

Function	Description
`take(n, lst)`	Takes first n elements
`drop(n, lst)`	Drops first n elements
`take_while(pred, lst)`	Takes while predicate is true
`drop_while(pred, lst)`	Drops while predicate is true
`zip(a, b)`	Zips two lists into pairs
`zip_with(f, a, b)`	Zips with a combining function
`unzip(lst)`	Unzips list of pairs

10.1.4 Aggregation

Function	Description
`sum(lst)`	Sum of all elements
`product(lst)`	Product of all elements
`maximum(lst)`	Maximum element
`minimum(lst)`	Minimum element
`count(pred, lst)`	Counts elements satisfying pred

10.2 Function Utilities

Function	Description
`compose(f, g)`	Returns $f \circ g$ (i.e., $x \mapsto f(g(x))$)
`pipe(f, g)`	Returns $g \circ f$ (i.e., $x \mapsto g(f(x))$)
`identity(x)`	Returns x unchanged
`constant(x)`	Returns function that always returns x
`flip(f)`	Flips argument order of binary function

10.3 I/O Functions

Function	Description
`print(x)`	Prints x without newline
`println(x)`	Prints x with newline
`input()`	Reads a line from stdin
`input_prompt(msg)`	Displays prompt, reads line

10.4 Type Conversion

Function	Description
`str(x)`	Converts to string
`int(x)`	Converts to integer
`float(x)`	Converts to float

10.5 Type Checking

Function	Description
`is_int(x)`	True if x is an integer
`is_float(x)`	True if x is a float
`is_string(x)`	True if x is a string
`is_bool(x)`	True if x is a boolean
`is_list(x)`	True if x is a list
`is_tuple(x)`	True if x is a tuple
`is_record(x)`	True if x is a record
`is_fn(x)`	True if x is a function
`type(x)`	Returns the type name as string

10.6 Math Functions

Function	Description
`abs(x)`	Absolute value
`sqrt(x)`	Square root
`pow(base, exp)`	Exponentiation
`min(a, b)`	Minimum of two values
`max(a, b)`	Maximum of two values
`floor(x)`	Floor (round down)
`ceil(x)`	Ceiling (round up)
`round(x)`	Round to nearest integer
`sin(x)`, `cos(x)`, `tan(x)`	Trigonometric functions
`log(x)`	Natural logarithm
`exp(x)`	Exponential (e^x)
`random()`	Random float in $[0, 1)$
`pi`	Mathematical constant π
`e`	Mathematical constant e

10.7 String Functions

Function	Description
`substr(s, start, len)`	Extracts substring
`split(s, delim)`	Splits string by delimiter
`join(lst, delim)`	Joins list with delimiter
`uppercase(s)`	Converts to uppercase
`lowercase(s)`	Converts to lowercase
`trim(s)`	Removes leading/trailing whitespace
`contains(s, sub)`	True if s contains sub
`starts_with(s, pre)`	True if s starts with pre
`ends_with(s, suf)`	True if s ends with suf
`replace(s, old, new)`	Replaces occurrences
`chars(s)`	Converts to list of characters
`index_of(s, sub)`	Index of first occurrence

10.8 Utility Functions

Function	Description
`error(msg)`	Raises an error with message
`assert(cond)`	Asserts condition is true
`compare(a, b)`	Returns -1, 0, or 1

11 Example Programs

11.1 Fibonacci Sequence

```
// Naive recursive implementation
fn fib(n) {
    match n {
        0 => 0,
```

```
        1 => 1,
        _ => fib(n - 1) + fib(n - 2)
    }
}

// Efficient tail-recursive implementation
fn fib_fast(n) {
    fn helper(n, a, b) {
        match n {
            0 => a,
            _ => helper(n - 1, b, a + b)
        }
    }
    helper(n, 0, 1)
}

println(fib_fast(20))  // 6765
```

11.2 QuickSort

```
fn quicksort(lst) {
    match lst {
        [] => [],
        [pivot, ...rest] => {
            let smaller = filter((x) => x < pivot, rest)
            let larger = filter((x) => x >= pivot, rest)
            append(
                append(quicksort(smaller), [pivot]),
                quicksort(larger)
            )
        }
    }
}

let sorted = quicksort([3, 1, 4, 1, 5, 9, 2, 6])
println(sorted)  // [1, 1, 2, 3, 4, 5, 6, 9]
```

11.3 Binary Tree Operations

```
type Tree {
    Leaf(value),
    Node(left, right)
}

fn tree_height(tree) {
    match tree {
        Leaf(_) => 1,
        Node(l, r) => 1 + max(tree_height(l), tree_height(r))
    }
}

fn tree_flatten(tree) {
    match tree {
        Leaf(x) => [x],
        Node(l, r) => append(tree_flatten(l), tree_flatten(r))
```

```
    }
}

fn tree_map(f, tree) {
    match tree {
        Leaf(x) => Leaf(f(x)),
        Node(l, r) => Node(tree_map(f, l), tree_map(f, r))
    }
}

let tree = Node(
    Node(Leaf(1), Leaf(2)),
    Node(Leaf(3), Node(Leaf(4), Leaf(5)))
)

println(tree_height(tree))    // 4
println(tree_flatten(tree))   // [1, 2, 3, 4, 5]
```

11.4 Option Monad

```
type Option {
    Some(value),
    None
}

fn map_opt(f, opt) {
    match opt {
        Some(x) => Some(f(x)),
        None => None
    }
}

fn flat_map_opt(f, opt) {
    match opt {
        Some(x) => f(x),
        None => None
    }
}

fn filter_opt(pred, opt) {
    match opt {
        Some(x) if pred(x) => Some(x),
        _ => None
    }
}

// Safe division
fn safe_div(a, b) {
    if b == 0 { None }
    else { Some(a / b) }
}

// Chaining operations
let result = flat_map_opt(
    (x) => safe_div(100, x),
    safe_div(20, 4)
)
```

```
// safe_div(20, 4) = Some(5)
// safe_div(100, 5) = Some(20)
// result = Some(20)
```

11.5 Functional Data Pipeline

```
// Process a list of numbers
let numbers = range(1, 101)  // [1, 2, ..., 100]

let result = numbers
    |> filter((x) => x % 2 == 0)       // Keep even numbers
    |> map((x) => x * x)                // Square them
    |> filter((x) => x > 100)           // Keep > 100
    |> take(5)                          // First 5

// Equivalent using composition
let process = compose(
    (lst) => take(5, lst),
    compose(
        (lst) => filter((x) => x > 100, lst),
        compose(
            (lst) => map((x) => x * x, lst),
            (lst) => filter((x) => x % 2 == 0, lst)
        )
    )
)

println(process(range(1, 101)))
// [144, 196, 256, 324, 400]
```

12 Conclusion

Setsuna is a thoughtfully designed functional programming language that brings together:

- **Powerful pattern matching** with support for literals, variables, wildcards, destructuring, and guards
- **Algebraic data types** for modeling complex domains safely
- **First-class functions** with closures and higher-order programming
- **Automatic type inference** eliminating boilerplate while maintaining safety
- **Immutability by default** encouraging pure, predictable code
- **Familiar C-style syntax** making it accessible to programmers from various backgrounds

The language is well-suited for applications requiring:

- Data transformation and processing pipelines
- Domain modeling with algebraic data types
- Algorithm implementation with recursive patterns
- Teaching functional programming concepts

For more information and examples, visit the official documentation and example files in the `examples/` and `docs/` directories of the Setsuna distribution.