STARK Core Language Specification Overview

Introduction

This document provides an overview of the complete STARK core language specification. The documents in `docs/spec/` are normative for Core v1. The core language defines the general-purpose language surface (lexing, syntax, types, semantics, memory, modules, and standard library). Non-core extensions are defined separately.

Design Philosophy

Core Principles

1. **Memory Safety**: Prevent common memory errors through ownership and borrowing
2. **Performance**: Zero-cost abstractions and predictable execution
3. **Clarity**: Simple, explicit syntax and semantics
4. **Pragmatism**: A minimal, implementable Core v1
5. **Interoperability**: Clear boundaries for future extensions

Language Goals

- A safe, compiled, general-purpose language core
- Compile-time guarantees for memory and type safety
- Simple, predictable semantics suitable for tooling
- Clear extension points for domain-specific features

Specification Structure

1. Lexical Grammar (01-Lexical-Grammar.md)

Defines how source code is tokenized: - **Keywords**: Control flow, declarations, types, operators - **Identifiers**: Variables, functions, types (snake_case/PascalCase) - **Literals**: Integers, floats, strings, characters, booleans - **Operators**: Arithmetic, comparison, logical, bitwise, assignment - **Comments**: Single-line (`//`) and multi-line (`/* */`) - **Whitespace**: Space, tab, newline handling

2. Syntax Grammar (02-Syntax-Grammar.md)

Defines the concrete syntax using EBNF: - **Program Structure**: Items (functions, structs, enums, traits) - **Expressions**: Precedence, associativity - **Statements**: Variable declarations, control flow, returns - **Type Syntax**: Primitives, composites, references, functions - **Pattern Matching**: Destructuring and exhaustiveness

3. Type System (03-Type-System.md)

Comprehensive type system with safety guarantees: - **Primitive Types**: Integers, floats, booleans, characters, strings - **Composite Types**: Arrays, tuples, structs, enums - **Reference Types**: Immutable and mutable references - **Ownership Model**: Move semantics, borrowing rules - **Type Inference**: Local and function return type inference - **Trait System**: Interfaces and generic constraints

4. Semantic Analysis (04-Semantic-Analysis.md)

Rules for meaningful program validation: - **Symbol Resolution**: Scoping, shadowing, name lookup - **Type Checking**: Assignment compatibility, function calls - **Ownership Analysis**: Move tracking, borrow checking - **Control Flow**: Reachability, return path analysis - **Pattern Exhaustiveness**: Match completeness checking - **Error Reporting**: Comprehensive diagnostics with suggestions

5. Memory Model (05-Memory-Model.md)

Memory safety through compile-time analysis: - **Ownership Rules**: Single ownership, automatic cleanup - **Move Semantics**: Explicit ownership transfer - **Borrowing System**: Immutable and mutable references - **Lifetime Tracking**: Reference validity guarantees - **Stack vs Heap**: Allocation strategy and layout - **Drop System**: Automatic and manual resource cleanup

6. Standard Library (06-Standard-Library.md)

Essential types and functions for practical programming: - **Core Types**: Option, Result, Box, Vec, HashMap - **String Handling**: Unicode-aware string operations - **Collections**: Dynamic arrays, hash tables, sets - **IO Operations**: File handling, console output - **Math Functions**: Arithmetic, trigonometric, random numbers - **Error Handling**: Structured error types and propagation

7. Modules and Packages (07-Modules-and-Packages.md)

Defines module structure, visibility, and import resolution: - **Modules**: File and directory layout - **Visibility**: `pub`/`priv` rules - **Imports**: `use` paths, trees, and aliasing - **Packages**: Manifest and dependency resolution

8. Non-Core Extensions (../extensions/AI-Extensions.md)

Non-core language extensions live outside Core v1 and are optional. See `docs/extensions/AI-Extensions.md`.

Core Language Features

Variables and Mutability

```
let x = 42              // Immutable by default
let mut y = 10          // Explicitly mutable
const MAX_SIZE: Int32 = 1000  // Compile-time constant
```

Functions

```
fn add(a: Int32, b: Int32) -> Int32 {
    a + b
}

fn greet(name: &str) {
    println("Hello, " + name)
}
```

Ownership and Borrowing

```
fn consume(s: String) {
    // s is owned here
}

fn borrow(s: &String) -> Int32 {
    s.len()
}

fn mutate(s: &mut String) {
    s.push('!')
}
```

Implementation Phases

Phase 1: Core MVP

Goal: A complete, implementable Core v1 - Lexer and parser for core syntax - Type checker with ownership analysis - Module system and import resolution - Minimal standard library

Phase 2: Tooling and Stability

Goal: A stable core suitable for real use - Improved diagnostics and error recovery - Formatter and basic tooling support - Expanded standard library coverage

Success Criteria

Correctness

- [ ] Memory safety enforced by ownership and borrowing
- [ ] Deterministic type checking and inference
- [ ] Exhaustive match checking

Developer Experience

- [ ] Clear, actionable error messages
- [ ] Predictable module and import rules
- [ ] Stable core language surface

Next Steps

1. **Finalize Core Grammar**: Resolve remaining ambiguities in syntax and lexing
2. **Solidify Type Rules**: Confirm inference and trait constraints
3. **Define Module Rules**: Ensure deterministic resolution and visibility
4. **Validate Stdlib Surface**: Confirm minimal APIs and behaviors

This specification provides a focused foundation for implementing a safe, performant, general-purpose language core.

Conformance

A conforming Core v1 implementation MUST follow the requirements in this document. Any deviations or extensions MUST be explicitly documented by the implementation.

STARK Lexical Grammar Specification

Overview

This document defines the lexical structure of STARK - how source code is broken down into tokens.

Token Categories

1. Keywords

```
// Control Flow
if, else, match
```

```
for, while, loop, break, continue, return

// Declarations
fn, struct, enum, trait, impl, let, mut, const, type, use, mod

// Types
Int8, Int16, Int32, Int64, UInt8, UInt16, UInt32, UInt64
Float32, Float64, Bool, String, Char, Unit, str

// Visibility
pub, priv

// Module Paths
self, super, crate

// Operators
and, or, not, in, is, as

// Literals
true, false
```

2. Identifiers

```
IDENTIFIER := [a-zA-Z_][a-zA-Z0-9_]*
```

Rules: - Must start with letter or underscore - Can contain letters, digits, underscores - Case sensitive - Cannot be keywords - Maximum length: 255 characters

3. Literals

Integer Literals

```
DECIMAL_INT := [0-9][0-9_]*
HEX_INT     := 0[xX][0-9a-fA-F][0-9a-fA-F_]*
BINARY_INT  := 0[bB][01][01_]*
OCTAL_INT   := 0[oO][0-7][0-7_]*

// Type suffixes
INT_SUFFIX := (i8|i16|i32|i64|u8|u16|u32|u64)
```

Examples:

```
42
1_000_000
0xFF_FF
0b1010_1010
0o755
42i32
255u8
```

Floating Point Literals

```
FLOAT := DECIMAL_INT '.' [0-9][0-9_]* [EXPONENT]?
       | DECIMAL_INT EXPONENT
EXPONENT := [eE][+-]?[0-9][0-9_]*

// Type suffixes
FLOAT_SUFFIX := (f32|f64)
```

Examples:

```
3.14
1.0e10
2.5e-3
42.0f32
```

String Literals

```
STRING := '"' (CHAR | ESCAPE_SEQUENCE)* '"'
RAW_STRING := 'r"' .*? '"'

ESCAPE_SEQUENCE := '\' (n|t|r|0|\\|'|"|x[0-9a-fA-F]{2}|u{[0-9a-fA-F]
{1,6}})
```

Examples:

```
"Hello, World!"
"Line 1\nLine 2"
"\x41\x42\x43"
"\u{1F600}"
r"Raw string with \n literal backslashes"
```

Character Literals

```
CHAR := '\'' (CHAR_CONTENT | ESCAPE_SEQUENCE) '\''
CHAR_CONTENT := [^'\\]
```

Examples:

```
'a'
'\n'
'\x41'
'\u{1F600}'
```

Boolean Literals

```
BOOL := true | false
```

4. Operators

Arithmetic

```
+ - * / % **
```

Comparison

```
== != < <= > >=
```

Logical

```
&& || !
```

Bitwise

```
& | ^ ~ << >>
```

Assignment

```
= += -= *= /= %= **= &= |= ^= <<= >>=
```

Other

```
? : :: . -> => @ # $
```

5. Delimiters

```
( ) [ ] { } , ; :
```

6. Comments

```
// Single line comment
/* Multi-line comment */
/** Documentation comment */
```

Rules: - Single line comments start with `//` and continue to end of line - Multi-line comments start with `/*` and end with `*/` - Multi-line comments can be nested - Documentation comments start with `/**` and are associated with following declaration

7. Whitespace

- Space (U+0020)
- Tab (U+0009)
- Newline (U+000A)
- Carriage Return (U+000D)

Whitespace separates tokens and is otherwise ignored. Statement termination is defined by semicolons in the syntax grammar.

8. Token Precedence

When multiple token patterns could match: 1. Keywords take precedence over identifiers 2. Longer operators take precedence over shorter ones 3. Comments are ignored in token stream

9. Reserved Tokens

Reserved for future use:

```
async, await, yield, where, macro, unsafe, extern, import, export,
null
```

Lexical Analysis Rules

1. **Maximal Munch**: Always match the longest possible token
2. **Whitespace**: Ignored except for token separation
3. **Encoding**: Source files must be valid UTF-8

Error Handling

Invalid tokens should produce specific error messages: - Invalid character: “Unexpected character ‘X’ at line Y:Z” - Unterminated string: “Unterminated string literal at line Y:Z” - Invalid number: “Invalid number format at line Y:Z” ## Conformance A conforming Core v1 implementation MUST follow the requirements in this document. Any deviations or extensions MUST be explicitly documented by the implementation.

STARK Syntax Grammar Specification

Overview

This document defines the concrete syntax grammar for STARK using Extended Backus-Naur Form (EBNF).

Grammar Notation

```
::= means "is defined as"
|   means "or" (alternative)
?   means "optional" (zero or one)
*   means "zero or more"
+   means "one or more"
()  means "grouping"
```

```
[]  means "optional"
{}  means "zero or more repetitions"
```

Top-Level Grammar

Program

```
Program ::= Item*

Item ::= Visibility? (Function
       | Struct
       | Enum
       | Trait
       | Impl
       | Const
       | TypeAlias
       | Use
       | Module)

Visibility ::= 'pub' | 'priv'
```

Function Definition

```
Function ::= 'fn' IDENTIFIER '(' ParameterList? ')' ('->' Type)? Block

ParameterList ::= Parameter (',' Parameter)* ','?

Parameter ::= IDENTIFIER ':' Type
            | 'mut' IDENTIFIER ':' Type

Block ::= '{' Statement* '}'
```

Data Type Definitions

```
Struct ::= 'struct' IDENTIFIER '{' FieldList? '}'

FieldList ::= Field (',' Field)* ','?

Field ::= IDENTIFIER ':' Type
        | 'pub' IDENTIFIER ':' Type

Enum ::= 'enum' IDENTIFIER '{' VariantList? '}'

VariantList ::= Variant (',' Variant)* ','?

Variant ::= IDENTIFIER
          | IDENTIFIER '(' TypeList ')'
          | IDENTIFIER '{' FieldList '}'
```

```
Trait ::= 'trait' IDENTIFIER '{' TraitItem* '}'

TraitItem ::= Function
            | Type

Impl ::= 'impl' Type '{' ImplItem* '}'
       | 'impl' IDENTIFIER 'for' Type '{' ImplItem* '}'

ImplItem ::= Function
```

Module Definition

```
Module ::= 'mod' IDENTIFIER ';'
        | 'mod' IDENTIFIER ModuleBlock

ModuleBlock ::= '{' Item* '}'
```

Type Alias

```
TypeAlias ::= 'type' IDENTIFIER '=' Type ';'
```

Statements

```
Statement ::= ';'                    // Empty statement
            | Expression ';'         // Expression statement
            | 'let' LetStatement
            | 'return' Expression? ';'
            | 'break' Expression? ';'
            | 'continue' ';'
            | Block

LetStatement ::= IDENTIFIER ':' Type ';'
               | 'mut' IDENTIFIER ':' Type ';'
               | IDENTIFIER ':' Type '=' Expression ';'
               | IDENTIFIER '=' Expression ';'
               | 'mut' IDENTIFIER ':' Type '=' Expression ';'
               | 'mut' IDENTIFIER '=' Expression ';'
```

Expressions

```
Expression ::= AssignmentExpression

AssignmentExpression ::= LogicalOrExpression
                       | LogicalOrExpression AssignOp
AssignmentExpression

AssignOp ::= '=' | '+=' | '-=' | '*=' | '/=' | '%=' | '**='
           | '&=' | '|=' | '^=' | '<<=' | '>>='
```

```
LogicalOrExpression ::= LogicalAndExpression ('||'
LogicalAndExpression)*

LogicalAndExpression ::= EqualityExpression ('&&' EqualityExpression)*

EqualityExpression ::= RelationalExpression (('==' | '!=')
RelationalExpression)*

RelationalExpression ::= BitwiseOrExpression (('<' | '<=' | '>' |
'>=') BitwiseOrExpression)*

BitwiseOrExpression ::= BitwiseXorExpression ('|'
BitwiseXorExpression)*

BitwiseXorExpression ::= BitwiseAndExpression ('^'
BitwiseAndExpression)*

BitwiseAndExpression ::= ShiftExpression ('&' ShiftExpression)*

ShiftExpression ::= AdditiveExpression (('<<' | '>>')
AdditiveExpression)*

AdditiveExpression ::= MultiplicativeExpression (('+' | '-')
MultiplicativeExpression)*

MultiplicativeExpression ::= ExponentiationExpression (('*' | '/' |
'%') ExponentiationExpression)*

ExponentiationExpression ::= UnaryExpression ('**'
ExponentiationExpression)?

UnaryExpression ::= PostfixExpression
                  | '-' UnaryExpression
                  | '!' UnaryExpression
                  | '~' UnaryExpression
                  | '&' UnaryExpression        // Reference
                  | '*' UnaryExpression        // Dereference

PostfixExpression ::= PrimaryExpression
                    | PostfixExpression '(' ArgumentList? ')'     //
Function call
                    | PostfixExpression '[' Expression ']'        //
Array access
                    | PostfixExpression '.' IDENTIFIER            //
Field access
                    | PostfixExpression '.' INTEGER               //
Tuple access
                    | PostfixExpression '?'                       //
Try operator
```

```
ArgumentList ::= Expression (',' Expression)∗ ','?

PrimaryExpression ::= IDENTIFIER
                    | Literal
                    | '(' Expression ')'
                    | ArrayLiteral
                    | StructLiteral
                    | IfExpression
                    | MatchExpression
                    | LoopExpression
                    | Block
```

Control Flow Expressions

```
IfExpression ::= 'if' Expression Block ('else' 'if' Expression Block)∗
('else' Block)?

MatchExpression ::= 'match' Expression '{' MatchArm∗ '}'

MatchArm ::= Pattern '=>' Expression ','?

Pattern ::= IDENTIFIER
          | Literal
          | '_'                                  // Wildcard
          | IDENTIFIER '(' PatternList? ')'      // Enum variant
          | '(' PatternList? ')'                 // Tuple
          | '[' PatternList? ']'                 // Array

PatternList ::= Pattern (',' Pattern)∗ ','?

LoopExpression ::= 'loop' Block
                 | 'while' Expression Block
                 | 'for' IDENTIFIER 'in' Expression Block
```

Literals

```
Literal ::= INTEGER
          | FLOAT
          | STRING
          | CHAR
          | BOOLEAN

ArrayLiteral ::= '[' ExpressionList? ']'

ExpressionList ::= Expression (',' Expression)∗ ','?

StructLiteral ::= IDENTIFIER '{' FieldInitList? '}'

FieldInitList ::= FieldInit (',' FieldInit)∗ ','?
```

```
FieldInit ::= IDENTIFIER ':' Expression
            | IDENTIFIER                   // Shorthand: field: field
```

Types

```
Type ::= PrimitiveType
       | IDENTIFIER                          // Named type
       | '[' Type ';' INTEGER ']'            // Array type
       | '[' Type ']'                        // Slice type
       | '(' TypeList? ')'                   // Tuple type
       | '&' Type                            // Reference type
       | '&' 'mut' Type                      // Mutable reference type
       | Type '->' Type                      // Function type

PrimitiveType ::= 'Int8' | 'Int16' | 'Int32' | 'Int64'
                | 'UInt8' | 'UInt16' | 'UInt32' | 'UInt64'
                | 'Float32' | 'Float64'
                | 'Bool' | 'Char' | 'String' | 'Unit' | 'str'

TypeList ::= Type (',' Type)* ','?
```

Other Constructs

```
Const ::= 'const' IDENTIFIER ':' Type '=' Expression ';'

Use ::= 'use' UseTree ';'

UseTree ::= Path ('as' IDENTIFIER)?
          | Path '::' '*'
          | Path '::' 'self'
          | Path '::' '{' UseTreeList? '}'

UseTreeList ::= UseTree (',' UseTree)* ','?

Path ::= PathSegment ('::' PathSegment)*
PathSegment ::= IDENTIFIER | 'self' | 'super' | 'crate'
```

Operator Precedence (Highest to Lowest)

1. Primary expressions, field access, array access, function calls, try operator (?)
2. Unary operators (-, !, ~, &, *)
3. Exponentiation (**)
4. Multiplicative (*, /, %)
5. Additive (+, -)
6. Shift (<<, >>)
7. Bitwise AND (&)
8. Bitwise XOR (^)
9. Bitwise OR (|)
10. Relational (<, <=, >, >=)

11. Equality (==, !=)
12. Logical AND (&&)
13. Logical OR (||)
14. Assignment (=, +=, -=, etc.)

Associativity

- Left associative: Most binary operators
- Right associative: Assignment operators, exponentiation
- Non-associative: Comparison operators

Statement vs Expression

- Statements do not return values and end with semicolons
- Expressions return values and can be used as statements with semicolons
- Blocks are expressions (return value of last expression, or Unit if none)
- Control flow constructs are expressions

Whitespace and Semicolons

- Semicolons are required to terminate statements
- Semicolons are optional after the last expression in a block
- Newlines are not significant except for line comments
- Trailing commas are allowed in lists

Grammar Extensions for Future Features

Reserved grammar constructs for later implementation:

```
// Generic types (future)
GenericType ::= IDENTIFIER '<' TypeList '>'

// Async functions (future)
AsyncFunction ::= 'async' 'fn' IDENTIFIER '(' ParameterList? ')' ('->'
Type)? Block

// Lambda expressions (future)
Lambda ::= '|' ParameterList? '|' (Expression | Block)
```

Conformance

A conforming Core v1 implementation MUST follow the requirements in this document. Any deviations or extensions MUST be explicitly documented by the implementation.

STARK Type System Specification

Overview

STARK features a static type system with type inference, ownership semantics, and memory safety guarantees.

Core Principles

1. **Static Typing**: All types resolved at compile time
2. **Type Inference**: Types can be inferred when unambiguous
3. **Memory Safety**: No null pointer dereferences, no use-after-free
4. **Ownership**: Clear ownership and borrowing rules
5. **Zero-cost Abstractions**: Type safety without runtime overhead

Primitive Types

Integer Types

```
Int8    // 8-bit signed integer (-128 to 127)
Int16   // 16-bit signed integer (-32,768 to 32,767)
Int32   // 32-bit signed integer (-2^31 to 2^31-1)
Int64   // 64-bit signed integer (-2^63 to 2^63-1)

UInt8   // 8-bit unsigned integer (0 to 255)
UInt16  // 16-bit unsigned integer (0 to 65,535)
UInt32  // 32-bit unsigned integer (0 to 2^32-1)
UInt64  // 64-bit unsigned integer (0 to 2^64-1)
```

Default integer type is `Int32` for literals that fit, `Int64` otherwise.

Floating Point Types

```
Float32  // 32-bit IEEE 754 floating point
Float64  // 64-bit IEEE 754 floating point
```

Default floating point type is `Float64`.

Other Primitive Types

```
Bool    // Boolean: true or false
Char    // Unicode scalar value (32-bit)
Unit    // Unit type: () - represents no meaningful value
str     // String slice (unsized), typically used behind references:
&str
```

String Type

```
String  // UTF-8 encoded string, heap-allocated, growable
```

String Slice Type

```
// str is an unsized string slice type. It is used via references.
let s: &str = "hello"
let owned: String = String::from(s)
```

Composite Types

Array Types

```
[T; N]   // Fixed-size array of N elements of type T
[T]      // Dynamic array (slice) of elements of type T
```

Examples:

```
let fixed: [Int32; 5] = [1, 2, 3, 4, 5]
let dynamic: [Int32] = [1, 2, 3]
```

Tuple Types

```
()           // Empty tuple (Unit type)
(T,)         // Single-element tuple
(T1, T2)     // Two-element tuple
(T1, T2, T3) // Three-element tuple
// ... up to reasonable limit (e.g., 16 elements)
```

Examples:

```
let empty: () = ()
let single: (Int32,) = (42,)
let pair: (Int32, String) = (42, "hello")
```

Struct Types

```
struct Point {
    x: Float64,
    y: Float64
}

struct Person {
    name: String,
    age: Int32,
    pub email: String  // Public field
}
```

Enum Types

```
enum Color {
    Red,
    Green,
    Blue
}

enum Option<T> {
    Some(T),
    None
}

enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

Reference Types

```
&T       // Immutable reference to T
&mut T   // Mutable reference to T
```

Function Types

```
fn(T1, T2) -> R    // Function taking T1, T2 and returning R
fn() -> R          // Function taking no parameters, returning R
fn(T)              // Function taking T, returning Unit
```

Type Aliases

```
type Age = Int32
type Point2D = (Float64, Float64)
type ErrorCode = Int32
```

Ownership and Borrowing

Ownership Rules

1. Each value has exactly one owner
2. When the owner goes out of scope, the value is dropped
3. Values can be moved (ownership transfer) or borrowed (temporary access)

Move Semantics

```
let a = String::new("hello")
let b = a  // a is moved to b, a is no longer valid
// print(a)  // Error: use of moved value
```

Borrowing Rules

1. You can have either one mutable reference or any number of immutable references
2. References must always be valid (no dangling pointers)
3. References cannot outlive the data they refer to

```
fn borrow_immutable(s: &String) {
    // Can read but not modify
}

fn borrow_mutable(s: &mut String) {
    // Can read and modify
}

let mut text = String::new("hello")
borrow_immutable(&text)        // OK
borrow_mutable(&mut text)      // OK
// borrow_immutable(&text)     // Error: cannot borrow as immutable
while mutable borrow exists
```

Type Inference

Local Type Inference

```
let x = 42          // Inferred as Int32
let y = 3.14        // Inferred as Float64
let z = [1, 2, 3]   // Inferred as [Int32]
```

Function Return Type Inference

```
fn add(a: Int32, b: Int32) {  // Return type inferred as Int32
    a + b
}
```

Generic Type Inference

```
fn identity<T>(x: T) -> T {
    x
}

let result = identity(42)  // T inferred as Int32
```

Subtyping and Coercion

Numeric Coercions

No implicit numeric conversions. Explicit casting required:

```
let x: Int32 = 42
let y: Int64 = x as Int64  // Explicit cast required
```

Reference Coercions

```
&mut T -> &T        // Mutable reference to immutable reference
&T -> &T            // Same type (identity)
```

Array to Slice Coercion

```
&[T; N] -> &[T]     // Array reference to slice reference
&mut [T; N] -> &mut [T]  // Mutable array reference to mutable slice
reference
```

Trait System (Basic)

Trait Definition

```
trait Display {
    fn fmt(&self) -> String
}

trait Eq {
    fn eq(&self, other: &Self) -> Bool
}
```

Trait Implementation

```
impl Display for Int32 {
    fn fmt(&self) -> String {
        // Convert integer to string
    }
}

impl Eq for Point {
    fn eq(&self, other: &Point) -> Bool {
        self.x == other.x && self.y == other.y
    }
}
```

Type Checking Rules

Assignment Compatibility

```
let x: T = expr  // expr must have type T or be coercible to T
```

Function Call Compatibility

```
fn f(param: T) { ... }
f(arg)  // arg must have type T or be coercible to T
```

Arithmetic Operations

```
// Binary arithmetic requires same types
let result = x + y  // x and y must have the same numeric type

// Comparison operators
let cmp = x < y     // x and y must have the same comparable type
```

Logical Operations

```
let result = a && b  // a and b must be Bool
let result = !x      // x must be Bool
```

Error Types and Handling

Option Type

```
enum Option<T> {
    Some(T),
    None
}
```

Result Type

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

Error Propagation

```
fn might_fail() -> Result<Int32, String> {
    // ...
}
```

```
fn caller() -> Result<Int32, String> {
    let value = might_fail()?  // Early return on error
    Ok(value * 2)
}
```

Try Operator (?) Typing

The try operator is defined for `Result<T, E>` and `Option<T>`: - If `expr` has type `Result<T, E>`, then `expr?` has type `T` and propagates `Err(E)` to the nearest enclosing function returning `Result<_, E>`. - If `expr` has type `Option<T>`, then `expr?` has type `T` and propagates `None` to the nearest enclosing function returning `Option<_>`.

The enclosing function's return type must be compatible with the propagated type.

Type System Extensions (Future)

Generics

```
struct Vec<T> {
    data: [T],
    len: Int32,
    cap: Int32
}

fn max<T: Ord>(a: T, b: T) -> T {
    if a > b { a } else { b }
}
```

Generics (Core v1)

Core v1 supports parametric polymorphism for functions, structs, enums, and traits.

Rules: - Generic parameters are introduced with `<T, U, ...>` after the item name. - Generic parameters are in scope within the item body and signatures. - All generic parameters used in an item MUST be declared by that item. - Instantiation occurs at use sites; Core v1 permits monomorphization or dictionary-passing, but the observable behavior MUST be equivalent.

Trait Bounds

```
fn max<T: Ord>(a: T, b: T) -> T { if a > b { a } else { b } }
```

Rules: - A bound `T: Trait` requires a visible `impl Trait for T`. - Multiple bounds are allowed: `T: TraitA + TraitB`.

Trait Coherence (Core v1)

To avoid ambiguous implementations, Core v1 applies the orphan rule: - An `impl` is valid only if either the trait or the type is defined in the current package.

Additionally: - If multiple applicable `impl` blocks could apply to the same type at a call site, the program is ill-formed. - Blanket implementations (e.g., `impl<T: Trait> OtherTrait for T`) are permitted but must not violate coherence.

Numeric Semantics (Core v1)

- Integer overflow and underflow are runtime errors and MUST trap.
- Division or modulo by zero is a runtime error and MUST trap.
- Floating-point operations follow IEEE-754 semantics (NaN, +/-Inf).

Associated Types

```
trait Iterator {
    type Item
    fn next(&mut self) -> Option<Self::Item>
}
```

Lifetime Parameters

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
```

Type Safety Guarantees

1. **No null pointer dereferences**: Option type prevents null access
2. **No buffer overflows**: Array bounds checking
3. **No use-after-free**: Ownership system prevents dangling pointers
4. **No data races**: Borrowing rules prevent concurrent access violations
5. **No memory leaks**: Automatic memory management through ownership

Implementation Notes

Type Representation

- Primitive types: Direct machine representation
- Composite types: Laid out according to platform ABI
- References: Pointers with compile-time tracking
- Enums: Tagged unions with optimal layout

Type Checking Algorithm

1. Parse source into AST
2. Build symbol table with declarations
3. Perform type inference using Hindley-Milner algorithm
4. Check type constraints and ownership rules
5. Generate type-annotated AST for code generation ## Conformance A conforming Core v1 implementation MUST follow the requirements in this document. Any deviations or extensions MUST be explicitly documented by the implementation.

STARK Semantic Analysis Specification

Overview

Semantic analysis validates that programs are meaningful according to STARK's language rules. This phase occurs after parsing and before code generation.

Analysis Phases

1. Symbol Table Construction

Build symbol tables for scopes and declarations.

Scope Rules

```
// Global scope
fn global_function() { }
const GLOBAL_CONST: Int32 = 42

fn example() {
    // Function scope
    let local_var = 10

    {
        // Block scope
        let block_var = 20
        // block_var accessible here
        // local_var accessible here
    }
    // block_var not accessible here
    // local_var still accessible
}
```

Name Resolution Order

1. Current scope
2. Parent scopes (innermost to outermost)
3. Global scope
4. Built-in names

Shadowing Rules

```
let x = 10        // x: Int32
{
    let x = "hi"  // Shadows outer x, x: String
    // Inner x takes precedence
}
// Outer x visible again
```

2. Type Checking

Variable Declarations

```
let x: Int32 = 42      // Type annotation matches literal
let y = 42             // Type inferred as Int32
let z: String = 42     // Error: type mismatch
```

Function Calls

```
fn add(a: Int32, b: Int32) -> Int32 { a + b }

let result = add(1, 2)     // OK: arguments match parameters
let error = add(1.0, 2)    // Error: Float64 cannot be Int32
let error2 = add(1)        // Error: wrong number of arguments
```

Assignment Compatibility

```
let mut x: Int32 = 10
x = 20                     // OK: same type
x = "hello"                // Error: type mismatch

let y: Int32 = 10
y = 20                     // Error: y is not mutable
```

3. Ownership and Borrowing Analysis

Move Semantics Validation

```
let s1 = String::new("hello")
let s2 = s1               // s1 moved to s2
print(s1)                 // Error: use of moved value
```

```
fn take_ownership(s: String) { }
let s3 = String::new("world")
take_ownership(s3)        // s3 moved into function
print(s3)                 // Error: use of moved value
```

Borrow Checking

```
let mut s = String::new("hello")
let r1 = &s               // Immutable borrow
let r2 = &s               // OK: multiple immutable borrows
let r3 = &mut s           // Error: cannot borrow mutably while
immutably borrowed

let mut s2 = String::new("world")
let r4 = &mut s2          // Mutable borrow
let r5 = &s2              // Error: cannot borrow immutably while
mutably borrowed
let r6 = &mut s2          // Error: cannot have multiple mutable
borrows
```

Lifetime Validation

```
fn invalid_return() -> &Int32 {
    let x = 42
    &x                    // Error: returning reference to local
variable
}

fn valid_return(x: &Int32) -> &Int32 {
    x                     // OK: returning input reference
}
```

4. Control Flow Analysis

Unreachable Code Detection

```
fn example() -> Int32 {
    return 42
    let x = 10            // Warning: unreachable code
}
```

Return Path Analysis

```
fn missing_return() -> Int32 {
    let x = 42
    // Error: not all paths return a value
}

fn conditional_return(flag: Bool) -> Int32 {
    if flag {
        return 1
    }
    // Error: missing return in else branch
}

fn valid_return(flag: Bool) -> Int32 {
    if flag {
        1
    } else {
        2
    }                      // OK: both branches return
}
```

Break/Continue Validation

```
fn invalid_break() {
    break                  // Error: break outside of loop
}

fn valid_break() {
    loop {
        break              // OK: break inside loop
    }
}
```

5. Pattern Matching Analysis

Exhaustiveness Checking

```
enum Color { Red, Green, Blue }

fn check_color(c: Color) -> String {
    match c {
        Color::Red => "red",
        Color::Green => "green"
        // Error: non-exhaustive pattern (missing Blue)
    }
}

fn valid_match(c: Color) -> String {
    match c {
        Color::Red => "red",
        Color::Green => "green",
        Color::Blue => "blue"     // OK: exhaustive
    }
}
```

Rules: - A `match` over an enum type is exhaustive if every variant is covered, or a wildcard (_) arm exists. - Tuple patterns are exhaustive if each element position is exhaustive for its type. - Literal patterns are exhaustive only for finite domains (e.g., `Bool` with `true` and `false`). - If a match is not exhaustive, it is a compile-time error.

Pattern Type Checking

```
let x: Int32 = 42
match x {
    "hello" => "string",          // Error: pattern type mismatch
    42 => "number"                // OK
}
```

6. Mutability Analysis

Mutable Access Validation

```
let x = 42
x = 43                    // Error: x is not mutable

let mut y = 42
y = 43                    // OK: y is mutable

struct Point { x: Int32, y: Int32 }
let p = Point { x: 1, y: 2 }
p.x = 10                  // Error: p is not mutable

let mut p2 = Point { x: 1, y: 2 }
p2.x = 10                 // OK: p2 is mutable
```

7. Initialization Analysis

Use Before Initialization

```
let x: Int32
print(x)                  // Error: use of uninitialized variable

let y: Int32 = if true { 42 } else { 24 }
print(y)                  // OK: y is initialized in all branches

let z: Int32
if condition {
    z = 42
}
print(z)                  // Error: z might not be initialized
```

Rules: - `let name: Type;` declares a variable without initializing it. - A variable must be definitely assigned before any read. - All control-flow paths must assign before use.

Double Initialization

```
let mut x: Int32 = 42
x = 43                    // OK: reassignment
let x = 44                // Error: redeclaration in same scope
```

8. Array Bounds Analysis

Static Bounds Checking

```
let arr: [Int32; 3] = [1, 2, 3]
let x = arr[2]            // OK: index in bounds
let y = arr[5]            // Error: index out of bounds (if
determinable)
```

Dynamic Bounds Checking

```
let arr: [Int32] = [1, 2, 3]
let idx = get_index()
let x = arr[idx]          // Runtime bounds check required
```

9. Error Propagation Analysis

Question Mark Operator

```
fn might_fail() -> Result<Int32, String> { ... }

fn caller1() -> Result<Int32, String> {
    let x = might_fail()?     // OK: compatible error types
    Ok(x * 2)
}

fn caller2() -> Int32 {
    let x = might_fail()?     // Error: function doesn't return Result
    x * 2
}
```

Rules: - `expr?` propagates `Err` or `None` to the nearest enclosing function returning `Result<_, E>` or `Option<_>`. - The enclosing function return type must be compatible with the propagated type.

Runtime Error Semantics (Core v1)

- A runtime error (e.g., integer overflow, division by zero, out-of-bounds indexing) MUST terminate the current program execution.
- `panic(...)` is a runtime error that terminates the program after emitting the provided message.

10. Trait Constraint Checking

Trait Bounds Validation

```
trait Display {
    fn fmt(&self) -> String
}

fn print_it<T: Display>(item: T) {
    print(item.fmt())         // OK: T implements Display
}

struct Point { x: Int32, y: Int32 }

print_it(Point { x: 1, y: 2 })  // Error: Point doesn't implement
Display
```

Error Reporting

Error Message Format

```
Error: [ERROR_CODE] [BRIEF_DESCRIPTION]
  --> file.stark:line:column
   |
line | source code line
   | ^^^^^ specific location
   |
   = help: detailed explanation
   = note: additional information
```

Error Categories

Type Errors (E0001-E0099)

- E0001: Type mismatch
- E0002: Unknown type
- E0003: Type annotation required
- E0004: Cannot infer type

Ownership Errors (E0100-E0199)

- E0100: Use of moved value
- E0101: Borrow check failed
- E0102: Lifetime violation
- E0103: Dangling reference

Name Resolution Errors (E0200-E0299)

- E0200: Undefined variable
- E0201: Undefined function
- E0202: Undefined type
- E0203: Ambiguous name

Control Flow Errors (E0300-E0399)

- E0300: Unreachable code
- E0301: Missing return value
- E0302: Break outside loop
- E0303: Non-exhaustive match

Warning Categories

Unused Items (W0001-W0099)

- W0001: Unused variable
- W0002: Unused function
- W0003: Unused import
- W0004: Dead code

Style Warnings (W0100-W0199)

- W0100: Non-snake_case variable
- W0101: Non-PascalCase type
- W0102: Missing documentation

Analysis Algorithm

Pass Order

1. **Declaration Pass**: Collect all top-level declarations
2. **Type Resolution Pass**: Resolve all type expressions
3. **Type Inference Pass**: Infer types for expressions
4. **Ownership Pass**: Check ownership and borrowing rules
5. **Control Flow Pass**: Analyze control flow and reachability
6. **Pattern Pass**: Check pattern exhaustiveness and types
7. **Constraint Pass**: Validate trait constraints and bounds

Dependency Resolution

- Forward references allowed for types and functions
- Circular dependencies detected and reported
- Initialization order determined for constants

Error Recovery

- Continue analysis after errors when possible
- Provide multiple related errors in single pass
- Suggest fixes when unambiguous
- Avoid cascading errors from single root cause

Implementation Considerations

Symbol Table Structure

```
struct SymbolTable {
    symbols: HashMap<String, Symbol>,
    parent: Option<&SymbolTable>,
    children: Vec<SymbolTable>
}

enum Symbol {
    Variable { ty: Type, mutable: bool, initialized: bool },
    Function { params: Vec<Type>, return_ty: Type },
    Type { definition: TypeDef },
    Constant { ty: Type, value: Value }
}
```

Type Checking Context

```
struct TypeContext {
    current_function: Option<FunctionId>,
    expected_return_type: Option<Type>,
    loop_depth: usize,
    borrowed_values: HashMap<ValueId, BorrowInfo>
}
```

Error Collection

```
struct ErrorReporter {
    errors: Vec<SemanticError>,
    warnings: Vec<SemanticWarning>,
    error_limit: usize
}
```

Conformance

A conforming Core v1 implementation MUST follow the requirements in this document. Any deviations or extensions MUST be explicitly documented by the implementation.

STARK Memory Model and Ownership Specification

Overview

STARK's memory model ensures memory safety without garbage collection through compile-time ownership tracking, similar to Rust but with some simplifications for the initial implementation.

Core Principles

1. Ownership

- Every value has exactly one owner
- When the owner goes out of scope, the value is dropped
- Ownership can be transferred (moved) but not duplicated

2. Borrowing

- Values can be borrowed without transferring ownership
- Immutable borrows allow reading
- Mutable borrows allow reading and writing
- Borrowing rules prevent data races at compile time

3. Lifetimes

- All references have a lifetime
- References cannot outlive the data they point to
- Lifetimes are mostly inferred by the compiler

Memory Layout

Stack Allocation

```
// Stack-allocated values
let x: Int32 = 42           // x stored on stack
let y: [Int32; 4] = [1,2,3,4]  // y stored on stack

struct Point { x: Float64, y: Float64 }
let p: Point = Point { x: 1.0, y: 2.0 }  // p stored on stack
```

Heap Allocation

```
// Heap-allocated values
let s: String = String::new("hello")     // String data on heap
let v: Vec<Int32> = Vec::new()          // Vec data on heap
let b: Box<Int32> = Box::new(42)        // Boxed value on heap
```

Reference Layout

```
// References are pointers (8 bytes on 64-bit)
let x: Int32 = 42
let r: &Int32 = &x          // r contains address of x

// Slices contain pointer + length
let arr: [Int32; 5] = [1,2,3,4,5]
let slice: &[Int32] = &arr[1..4]  // pointer + length (16 bytes)
```

Ownership Rules

Rule 1: Single Ownership

```
let s1 = String::new("hello")
let s2 = s1                 // Ownership moved from s1 to s2
// println(s1)              // Error: s1 no longer valid
println(s2)                 // OK: s2 owns the string
```

Rule 2: Automatic Cleanup

```
{
    let s = String::new("hello")  // s owns the string
    // String is automatically freed when s goes out of scope
}
```

Rule 3: Function Parameters

```
fn take_ownership(s: String) {
    // s is owned by this function
    println(s)
    // s is dropped when function returns
}

fn main() {
    let s = String::new("hello")
    take_ownership(s)       // Ownership transferred to function
    // println(s)           // Error: s no longer valid
}
```

Move Semantics

What Gets Moved

Types are moved if they: 1. Don’t implement the `Copy` trait 2. Contain non-Copy fields

```
// Types that are moved by default
String, Vec<T>, Box<T>, custom structs without Copy

// Types that are copied by default (implement Copy)
Int32, Float64, Bool, Char, &T, [T; N] where T: Copy
```

Move in Assignments

```
let s1 = String::new("hello")
let s2 = s1                 // Move
let s3 = s2.clone()         // Explicit copy (if Clone implemented)
```

Move in Function Calls

```
fn process(s: String) { ... }

let my_string = String::new("hello")
process(my_string)          // my_string moved into function
// my_string no longer accessible
```

Move in Returns

```
fn create_string() -> String {
    let s = String::new("hello")
    s                       // Ownership moved to caller
}
```

Borrowing System

Immutable Borrowing

```
fn read_string(s: &String) -> Int32 {
    s.len()                 // Can read but not modify
}

let my_string = String::new("hello")
let length = read_string(&my_string)  // Borrow my_string
println(my_string)          // my_string still accessible
```

Mutable Borrowing

```
fn modify_string(s: &mut String) {
    s.push('!')             // Can read and modify
}

let mut my_string = String::new("hello")
```

```
modify_string(&mut my_string)         // Mutable borrow
println(my_string)          // Prints "hello!"
```

Borrowing Rules

1. **Either** one mutable borrow **OR** any number of immutable borrows
2. References must always be valid

```
let mut s = String::new("hello")

// Multiple immutable borrows - OK
let r1 = &s
let r2 = &s
println("{} {}", r1, r2)

// Mutable and immutable borrow - Error
let r3 = &s
let r4 = &mut s             // Error: cannot borrow mutably while
immutably borrowed

// Multiple mutable borrows - Error
let r5 = &mut s
let r6 = &mut s             // Error: cannot borrow mutably twice
```

Lifetime System

Lifetime Basics

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
// Compiler infers that return value has same lifetime as input
parameters
```

Lifetime Annotations (Future Feature)

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
```

Dangling Reference Prevention

```
fn invalid() -> &Int32 {
    let x = 42
    &x                      // Error: returning reference to local
variable
}

fn valid(x: &Int32) -> &Int32 {
    x                       // OK: returning input reference
}
```

Memory Management Strategies

Stack vs Heap Decision

```
// Stack allocated (small, known size)
let point = Point { x: 1.0, y: 2.0 }
let array = [1, 2, 3, 4, 5]

// Heap allocated (dynamic size, large objects)
let string = String::new("hello")
let vector = Vec::new()
let boxed = Box::new(large_object)
```

Reference Counting (Rc/Arc)

```
// Single-threaded reference counting (future feature)
let data = Rc::new(vec![1, 2, 3])
let data2 = data.clone()    // Increment reference count

// Multi-threaded reference counting (future feature)
let shared = Arc::new(vec![1, 2, 3])
let shared2 = shared.clone()
```

Drop and Destructors

Automatic Drop

```
struct FileHandle {
    path: String,
    handle: Int32
}

impl Drop for FileHandle {
    fn drop(&mut self) {
        // Close file handle
        close_file(self.handle)
    }
}
```

```
{
    let file = FileHandle { path: "test.txt".to_string(), handle: 42 }
    // file.drop() called automatically when leaving scope
}
```

Drop Order

```
{
    let x = String::new("first")
    let y = String::new("second")
    let z = String::new("third")
    // Drop order: z, y, x (reverse declaration order)
}
```

Manual Drop

```
let s = String::new("hello")
drop(s)                     // Explicitly drop s
// println(s)               // Error: s has been dropped
```

Copy vs Move Types

Copy Types

Implement `Copy` trait - assignment creates a copy, not a move:

```
// Built-in Copy types
Int8, Int16, Int32, Int64, UInt8, UInt16, UInt32, UInt64
Float32, Float64, Bool, Char, Unit
&T (references), [T; N] where T: Copy

// Usage
let x: Int32 = 42
let y = x                   // x is copied, still accessible
println("{} {}", x, y)      // OK: both x and y valid
```

Move Types

Do not implement `Copy` - assignment moves ownership:

```
// Built-in Move types
String, Vec<T>, Box<T>, HashMap<K, V>

// Custom types are Move by default
struct Person {
    name: String,
    age: Int32
}
```

```
let p1 = Person { name: "Alice".to_string(), age: 30 }
let p2 = p1                 // p1 moved to p2
// println(p1.name)         // Error: p1 no longer valid
```

Smart Pointers

Box - Heap Allocation

```
let boxed_int = Box::new(42)
let large_array = Box::new([0; 1000])  // Allocate large array on heap
```

Rc - Reference Counting (Future)

```
let data = Rc::new(vec![1, 2, 3])
let reference1 = data.clone()    // Increment reference count
let reference2 = data.clone()    // Increment reference count
// Data deallocated when all references dropped
```

RefCell - Interior Mutability (Future)

```
let data = RefCell::new(42)
{
    let mut borrowed = data.borrow_mut()  // Runtime borrow check
    *borrowed = 43
}
println(data.borrow())      // Prints 43
```

Memory Safety Guarantees

What STARK Prevents

1. **Null pointer dereferences**: No null pointers, use Option
2. **Buffer overflows**: Array bounds checking
3. **Use after free**: Ownership system prevents dangling pointers
4. **Double free**: Ownership ensures single deallocation
5. **Memory leaks**: Automatic cleanup through ownership
6. **Data races**: Borrowing rules prevent concurrent access

Runtime Checks

Some checks remain at runtime: - Array bounds checking (unless optimized away) - Integer overflow (in debug mode) - RefCell borrow checking (future feature)

Performance Considerations

Zero-Cost Abstractions

- Ownership and borrowing have no runtime cost
- References are just pointers
- Move semantics avoid unnecessary copies

Optimization Opportunities

- Dead code elimination for unused values
- Lifetime optimization to reduce copies
- Stack allocation for escaped values when possible

Memory Layout Optimization

- Struct field reordering to minimize padding
- Enum layout optimization for tagged unions
- Array and slice bounds check elimination

Implementation Notes

Compiler Phases

1. **Ownership Analysis**: Track value ownership through program
2. **Borrow Checking**: Validate borrowing rules
3. **Lifetime Inference**: Determine reference lifetimes
4. **Drop Insertion**: Insert drop calls at scope exits
5. **Memory Layout**: Determine stack vs heap allocation

Error Messages

```
Error: borrow of moved value
  --> example.stark:10:5
   |
 8 |     let s1 = String::new("hello");
   |         -- move occurs because `s1` has type `String`
 9 |     let s2 = s1;
   |              -- value moved here
10 |     println(s1);
   |     ^^ value borrowed here after move
```

Integration with Type System

- Ownership information included in type signatures
- Borrowing constraints encoded in function types
- Lifetime parameters for generic functions (future)

Future Extensions

Advanced Features

- Lifetime parameters and annotations
- Higher-ranked trait bounds
- Associated types with lifetime parameters
- Async/await with proper lifetime handling ## Conformance A conforming Core v1 implementation MUST follow the requirements in this document. Any deviations or extensions MUST be explicitly documented by the implementation.

STARK Standard Library Specification

Overview

The STARK standard library provides essential types, functions, and modules for core programming tasks. This specification defines the minimal standard library for the initial implementation.

Core Module Structure

```
std/
├── core/           // Core language items
├── collections/    // Data structures
├── io/            // Input/output operations
├── string/        // String manipulation
├── math/          // Mathematical operations
├── mem/           // Memory management utilities
├── error/         // Error handling types
└── prelude/       // Automatically imported items
```

Prelude Module

Automatically imported into every STARK program:

```
// Basic types (no import needed)
Int8, Int16, Int32, Int64, UInt8, UInt16, UInt32, UInt64
Float32, Float64, Bool, Char, String, str, Unit

// Essential traits
trait Copy
trait Clone
trait Drop
trait Eq
trait Ord

// Essential types
enum Option<T> {
    Some(T),
    None
}

enum Result<T, E> {
    Ok(T),
    Err(E)
}

// Essential functions
fn print(value: &str)
fn println(value: &str)
fn panic(message: &str) -> !
```

Core Module (std::core)

Memory Management

```
// Box - heap allocation
struct Box<T> {
    ptr: *mut T
}

impl<T> Box<T> {
    fn new(value: T) -> Box<T>
    fn into_inner(self) -> T
}

// Manual memory management
fn drop<T>(value: T)
fn size_of<T>() -> UInt64
fn align_of<T>() -> UInt64
```

Option Type

```
enum Option<T> {
    Some(T),
    None
}

impl<T> Option<T> {
    fn is_some(&self) -> Bool
    fn is_none(&self) -> Bool
    fn unwrap(self) -> T
    fn unwrap_or(self, default: T) -> T
    fn map<U>(self, f: fn(T) -> U) -> Option<U>
    fn and_then<U>(self, f: fn(T) -> Option<U>) -> Option<U>
}
```

Result Type

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}

impl<T, E> Result<T, E> {
    fn is_ok(&self) -> Bool
    fn is_err(&self) -> Bool
    fn unwrap(self) -> T
    fn unwrap_or(self, default: T) -> T
    fn map<U>(self, f: fn(T) -> U) -> Result<U, E>
    fn map_err<F>(self, f: fn(E) -> F) -> Result<T, F>
    fn and_then<U>(self, f: fn(T) -> Result<U, E>) -> Result<U, E>
}
```

Collections Module (std::collections)

Vec - Dynamic Array

```
struct Vec<T> {
    ptr: *mut T,
    len: UInt64,
    cap: UInt64
}

impl<T> Vec<T> {
    fn new() -> Vec<T>
    fn with_capacity(capacity: UInt64) -> Vec<T>
    fn push(&mut self, item: T)
    fn pop(&mut self) -> Option<T>
    fn len(&self) -> UInt64
    fn capacity(&self) -> UInt64
    fn is_empty(&self) -> Bool
    fn get(&self, index: UInt64) -> Option<&T>
    fn get_mut(&mut self, index: UInt64) -> Option<&mut T>
    fn insert(&mut self, index: UInt64, item: T)
    fn remove(&mut self, index: UInt64) -> T
    fn clear(&mut self)
    fn append(&mut self, other: &mut Vec<T>)
    fn extend(&mut self, iter: impl Iterator<Item = T>)
}

// Index access
```

```
impl<T> Index<UInt64> for Vec<T> {
    type Output = T
    fn index(&self, index: UInt64) -> &T
}

impl<T> IndexMut<UInt64> for Vec<T> {
    fn index_mut(&mut self, index: UInt64) -> &mut T
}
```

HashMap<K, V> - Hash Table

```
struct HashMap<K, V> {
    // Internal implementation details
}

impl<K: Hash + Eq, V> HashMap<K, V> {
    fn new() -> HashMap<K, V>
    fn with_capacity(capacity: UInt64) -> HashMap<K, V>
    fn insert(&mut self, key: K, value: V) -> Option<V>
    fn get(&self, key: &K) -> Option<&V>
    fn get_mut(&mut self, key: &K) -> Option<&mut V>
    fn remove(&mut self, key: &K) -> Option<V>
    fn contains_key(&self, key: &K) -> Bool
    fn len(&self) -> UInt64
    fn is_empty(&self) -> Bool
    fn clear(&mut self)
    fn keys(&self) -> KeysIter<K>
    fn values(&self) -> ValuesIter<V>
    fn iter(&self) -> Iter<K, V>
}
```

HashSet - Hash Set

```
struct HashSet<T> {
    map: HashMap<T, Unit>
}

impl<T: Hash + Eq> HashSet<T> {
    fn new() -> HashSet<T>
    fn insert(&mut self, value: T) -> Bool
    fn remove(&mut self, value: &T) -> Bool
    fn contains(&self, value: &T) -> Bool
    fn len(&self) -> UInt64
    fn is_empty(&self) -> Bool
    fn clear(&mut self)
    fn iter(&self) -> Iter<T>
}
```

String Module (std::string)

String Type

```
struct String {
    bytes: Vec<UInt8>
}

impl String {
    fn new() -> String
    fn with_capacity(capacity: UInt64) -> String
    fn from(s: &str) -> String
    fn len(&self) -> UInt64
    fn is_empty(&self) -> Bool
    fn push(&mut self, ch: Char)
    fn push_str(&mut self, s: &str)
    fn pop(&mut self) -> Option<Char>
    fn clear(&mut self)
    fn chars(&self) -> CharsIter
    fn bytes(&self) -> &[UInt8]
    fn as_str(&self) -> &str
    fn into_bytes(self) -> Vec<UInt8>
    fn substring(&self, start: UInt64, end: UInt64) -> &str
    fn contains(&self, pattern: &str) -> Bool
    fn starts_with(&self, pattern: &str) -> Bool
    fn ends_with(&self, pattern: &str) -> Bool
    fn find(&self, pattern: &str) -> Option<UInt64>
    fn replace(&self, from: &str, to: &str) -> String
    fn split(&self, delimiter: &str) -> SplitIter
    fn trim(&self) -> &str
    fn to_lowercase(&self) -> String
    fn to_uppercase(&self) -> String
}

// String literals (&str)
impl str {
    fn len(&self) -> UInt64
    fn is_empty(&self) -> Bool
    fn chars(&self) -> CharsIter
    fn bytes(&self) -> &[UInt8]
    fn to_string(&self) -> String
    // ... similar methods to String
}
```

Math Module (std::math)

Basic Operations

```
// Constants
const PI: Float64 = 3.141592653589793
const E: Float64 = 2.718281828459045

// Basic functions
fn abs<T: Num>(x: T) -> T
fn min<T: Ord>(a: T, b: T) -> T
fn max<T: Ord>(a: T, b: T) -> T
fn clamp<T: Ord>(value: T, min: T, max: T) -> T

// Floating point functions
fn sqrt(x: Float64) -> Float64
fn pow(base: Float64, exp: Float64) -> Float64
fn log(x: Float64) -> Float64
fn log10(x: Float64) -> Float64
fn exp(x: Float64) -> Float64

// Trigonometric functions
fn sin(x: Float64) -> Float64
fn cos(x: Float64) -> Float64
fn tan(x: Float64) -> Float64
fn asin(x: Float64) -> Float64
fn acos(x: Float64) -> Float64
fn atan(x: Float64) -> Float64
fn atan2(y: Float64, x: Float64) -> Float64

// Rounding functions
fn floor(x: Float64) -> Float64
fn ceil(x: Float64) -> Float64
fn round(x: Float64) -> Float64
fn trunc(x: Float64) -> Float64

// Random numbers (simple linear congruential generator)
struct Random {
    seed: UInt64
}

impl Random {
    fn new(seed: UInt64) -> Random
    fn next_int(&mut self) -> UInt64
    fn next_float(&mut self) -> Float64
    fn range(&mut self, min: Int32, max: Int32) -> Int32
}
```

IO Module (std::io)

Basic IO Operations

```
// Standard streams
fn print(text: &str)
fn println(text: &str)
fn eprint(text: &str)     // stderr
fn eprintln(text: &str)   // stderr

// Simple file operations
struct File {
    handle: Int32
}

impl File {
    fn open(path: &str) -> Result<File, IOError>
    fn create(path: &str) -> Result<File, IOError>
    fn read_to_string(&mut self) -> Result<String, IOError>
    fn write(&mut self, data: &[UInt8]) -> Result<UInt64, IOError>
    fn write_str(&mut self, text: &str) -> Result<UInt64, IOError>
    fn close(self) -> Result<Unit, IOError>
}

// Error types
enum IOError {
    NotFound,
    PermissionDenied,
    AlreadyExists,
    InvalidInput,
    Other(String)
}

// Utility functions
fn read_file(path: &str) -> Result<String, IOError>
fn write_file(path: &str, content: &str) -> Result<Unit, IOError>
```

Error Module (std::error)

Error Trait

```
trait Error {
    fn message(&self) -> String
    fn source(&self) -> Option<&dyn Error>
}

// Standard error types
struct GenericError {
    message: String
}

impl Error for GenericError {
    fn message(&self) -> String {
        self.message.clone()
    }

    fn source(&self) -> Option<&dyn Error> {
        None
    }
}
```

Memory Module (std::mem)

Memory Utilities

```
// Memory operations
fn size_of<T>() -> UInt64
fn align_of<T>() -> UInt64
fn swap<T>(a: &mut T, b: &mut T)
fn replace<T>(dest: &mut T, src: T) -> T
fn take<T: Default>(dest: &mut T) -> T

// Unsafe memory operations (future)
fn copy<T>(src: *const T, dst: *mut T, count: UInt64)
fn copy_nonoverlapping<T>(src: *const T, dst: *mut T, count: UInt64)
```

Iterator Trait (std::iter)

Basic Iterator Interface

```
trait Iterator {
    type Item

    fn next(&mut self) -> Option<Self::Item>

    // Default implementations
    fn count(self) -> UInt64
    fn collect<C: FromIterator<Self::Item>>(self) -> C
    fn map<U>(self, f: fn(Self::Item) -> U) -> MapIter<Self, U>
    fn filter(self, predicate: fn(&Self::Item) -> Bool) ->
FilterIter<Self>
    fn fold<B>(self, init: B, f: fn(B, Self::Item) -> B) -> B
    fn reduce(self, f: fn(Self::Item, Self::Item) -> Self::Item) ->
Option<Self::Item>
    fn any(self, predicate: fn(Self::Item) -> Bool) -> Bool
    fn all(self, predicate: fn(Self::Item) -> Bool) -> Bool
    fn find(self, predicate: fn(&Self::Item) -> Bool) ->
Option<Self::Item>
}

trait FromIterator<T> {
    fn from_iter<I: Iterator<Item = T>>(iter: I) -> Self
}
```

Conversion Traits

Basic Conversion

```
trait From<T> {
    fn from(value: T) -> Self
}

trait Into<T> {
    fn into(self) -> T
}

trait TryFrom<T> {
    type Error
    fn try_from(value: T) -> Result<Self, Self::Error>
}

trait TryInto<T> {
    type Error
    fn try_into(self) -> Result<T, Self::Error>
}

// String conversion
trait ToString {
    fn to_string(&self) -> String
}

trait FromStr {
    type Error
    fn from_str(s: &str) -> Result<Self, Self::Error>
}
```

Essential Trait Implementations

Default Implementations

```
// Copy trait for basic types
impl Copy for Int8 { }
impl Copy for Int16 { }
impl Copy for Int32 { }
impl Copy for Int64 { }
impl Copy for UInt8 { }
impl Copy for UInt16 { }
impl Copy for UInt32 { }
impl Copy for UInt64 { }
impl Copy for Float32 { }
impl Copy for Float64 { }
impl Copy for Bool { }
impl Copy for Char { }
impl Copy for Unit { }

// Clone trait for all types
impl<T: Copy> Clone for T {
    fn clone(&self) -> T { *self }
}

// Eq trait for basic types
impl Eq for Int32 {
    fn eq(&self, other: &Int32) -> Bool { *self == *other }
}
// ... similar for other types

// Ord trait for basic types
impl Ord for Int32 {
    fn cmp(&self, other: &Int32) -> Ordering {
        if *self < *other { Ordering::Less }
        else if *self > *other { Ordering::Greater }
        else { Ordering::Equal }
    }
}
```

Implementation Priorities

Phase 1 (MVP)

- Basic types (primitives, String, Option, Result)
- Vec dynamic array
- Basic IO (print, println, simple file operations)
- Essential traits (Copy, Clone, Eq, Ord)

Phase 2 (Enhanced)

- HashMap<K, V> and HashSet
- Iterator trait and basic iterators
- Math module with common functions
- Enhanced string operations

Phase 3 (Complete)

- Advanced memory utilities
- Full IO system with buffering
- Regular expressions
- Time and date handling
- Thread and concurrency primitives (future)

Platform Considerations

Cross-platform Abstractions

- File path handling
- Directory operations
- Environment variables
- Process spawning (future)

Performance Notes

- Vec uses exponential growth strategy
- HashMap uses open addressing with Robin Hood hashing
- String operations are UTF-8 aware
- Iterator chains compile to efficient loops

Behavioral Requirements (Core v1)

- Indexing `Vec<T>` with `[]` MUST perform bounds checking and MUST trap on out-of-bounds access.
- `get`/`get_mut` MUST return `None` for out-of-bounds indices and MUST NOT trap.
- `String::substring(start, end)` MUST validate UTF-8 boundaries and MUST trap on invalid boundaries or ranges.
- IO functions MUST return `Result` with a non-`Ok` variant on failure and MUST NOT silently ignore errors. ## Conformance A conforming Core v1 implementation MUST follow the requirements in this document. Any deviations or extensions MUST be explicitly documented by the implementation.

STARK Modules and Packages Specification

Overview

This document defines the module system, visibility rules, and package resolution for the STARK core language. It is normative for Core v1.

Package Layout

A package is a directory containing a `starkpkg.json` manifest at its root.

- The manifest's `entry` field defines the root source file.
- If `entry` is omitted, the default is `src/main.stark`.

All module paths are resolved relative to the package root unless otherwise noted.

Module Declarations

Modules are declared with the `mod` keyword.

```
mod math;

mod inline {
    pub fn add(a: Int32, b: Int32) -> Int32 { a + b }
}
```

File-Based Modules

A declaration `mod name;` loads one of the following files, in order: 1. `name.stark` 2. `name/mod.stark`

The loaded file defines the contents of the module `name`.

Module Paths

Paths use `::` separators.

- `crate` refers to the package root module.
- `self` refers to the current module.
- `super` refers to the parent module.

Examples:

```
use crate::utils::math::add;
use super::config;
```

Imports (`use`)

The `use` statement brings names into scope.

```
use crate::utils::math::add;
use crate::utils::{math, io};
use crate::utils::math as m;
use crate::utils::*;
```

Rules: - `use` affects name resolution in the current module only. - `pub use` re-exports the imported name from the current module. - Aliases with `as` are local to the current module.

Visibility

- Items are private to their defining module by default.
- `pub` makes an item visible to parent modules and external modules.
- `priv` explicitly marks an item as private (same as default).

Visibility applies to: - Functions, structs, enums, traits, impl blocks, consts, type aliases, and modules.

Name Resolution Order

Within a module, names are resolved in the following order: 1. Local scope 2. Items declared in the current module 3. Items brought into scope by `use` 4. Items in parent modules via explicit `super::` paths 5. Items in `crate` via explicit `crate::` paths

Unqualified names do not implicitly search parent or crate scopes.

Packages and Dependencies

Dependencies are declared in `starkpkg.json` under `dependencies`.

Resolution order for external packages: 1. Local package source (the current package) 2. Direct dependencies from `starkpkg.json` 3. Standard library package `std`

Dependency modules are accessed via their package name:

```
use TensorLib::tensor::Tensor;
```

Dependency Version Resolution (Core v1)

- Version strings follow semantic versioning.
- If multiple versions satisfy a constraint, the highest version MUST be selected.
- If no version satisfies a constraint, compilation MUST fail with an error.
- The source of packages (registry, cache, or local path) is implementation-defined, but the chosen version MUST be reported in build output.

Standard Library

The standard library is available under the `std` package name:

```
use std::io;
use std::collections::Vec;
```

Cycles

Direct cyclic module dependencies are a compile-time error.

Errors

- Importing an unknown module or item is a compile-time error.
- Accessing a private item outside its module is a compile-time error.
- Ambiguous imports are a compile-time error unless aliased. ## Conformance A conforming Core v1 implementation MUST follow the requirements in this document. Any deviations or extensions MUST be explicitly documented by the implementation.
