HLang Programming Language Specification

Version 1.0 - June 2025

Table of Contents

Introduction

HLang (Hybrid Language) is a simplified programming language designed for educational purposes in compiler construction. Unlike traditional languages, HLang combines imperative and functional programming paradigms while maintaining simplicity for student implementation.

Key features of HLang:

- Static typing with limited type inference
- First-class functions
- Immutable data structures by default
- Simple but expressive syntax
- Memory-safe design

Program Structure

A HLang program consists of:

1. Optional constant declarations
2. Function declarations
3. A mandatory `main` function as the entry point

Example program structure:

```
const PI: float = 3.14159;

func factorial(n: int) -> int {
```

```
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}

func main() -> void {
    print("Factorial of 5 is: " + str(factorial(5)));
}
```

Lexical Structure

Character Set

HLang programs are written using the ASCII character set (7-bit encoding, characters 0-127). This design choice ensures maximum compatibility across different systems and simplifies the lexical analysis phase of compilation.

Supported Character Categories:

ASCII Letters: `A–Z`, `a–z` (codes 65-90, 97-122)

- Used for keywords, identifiers, and operators
- Case-sensitive throughout the language
- Form the basis of all language constructs

ASCII Digits: `0–9` (codes 48-57)

- Used in numeric literals (integers and floats)
- Used in identifiers (but not as the first character)
- Support for all decimal digits

Whitespace Characters: HLang recognizes the following ASCII whitespace characters for token separation:

Character	ASCII Code	Escape Sequence	Description
Space	32	`' '`	Standard space character
Horizontal Tab	9	`'\t'`	Tab character for indentation
Carriage Return	13	`'\r'`	Return character (line ending)
Newline	10	`'\n'`	Line feed character (line ending)

Line Ending Conventions:

- **Unix/Linux/macOS:** `\n` (LF)
- **Windows:** `\r\n` (CRLF)
- **Classic Mac:** `\r` (CR)
- All line ending conventions are automatically recognized and handled

Printable ASCII Characters: All printable ASCII characters (codes 32-126) are supported:

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

Special Characters in HLang: The following ASCII characters have special meanings:

- **Operators:** `+ - * / % == != < <= > >= && || ! = >>`
- **Separators:** `{}, ; . :`
- **String delimiters:** `"`
- **Comment markers:** `// /* */`

Character Classification Examples:

```
// Valid ASCII identifiers
let myVariable = 42;
let _internal = true;
let counter123 = 0;

// ASCII-only string literals
let greeting = "Hello, World!";
let symbols = "Special chars: !@#$%^&*()";

// ASCII-only comments
// This is a valid ASCII comment
/* Block comment with ASCII characters only */

// Error: Non-ASCII characters not supported
// let café = "Invalid";  // Compile error: non-ASCII in identifier
// let msg = "Café";      // Compile error: non-ASCII in string
```

Source File Requirements:

- Source files must contain only ASCII characters (0-127)
- Files should be saved with ASCII or UTF-8 encoding (UTF-8 for ASCII compatibility)
- BOM (Byte Order Mark) is not required and should be avoided
- Files containing characters with codes > 127 will be rejected during lexical analysis

Comments

HLang provides comprehensive commenting capabilities to support code documentation and temporary code disabling. Comments are completely ignored during lexical analysis and do not affect program execution.

Line Comments (`//`):

Line comments begin with two forward slashes (`//`) and continue until the end of the current line. They are ideal for brief explanations and end-of-line documentation.

Syntax: `// comment text`

Characteristics:

- Start with `//` (two consecutive forward slashes)
- Continue until the end of the current line
- Line ending terminates the comment (any of `\n`, `\r\n`, or `\r`)
- Can appear on their own line or at the end of a code line
- Cannot span multiple lines
- Nested `//` has no special meaning within line comments

Examples:

```
// This is a complete line comment
let x = 42;        // End-of-line comment explaining variable

// Multiple single-line comments
// can be used to create
// multi-line documentation

func factorial(n: int) -> int {
    // Base case: factorial of 0 or 1 is 1
    if (n <= 1) {
        return 1;   // Return base case value
    }
    // Recursive case: n! = n * (n-1)!
    return n * factorial(n - 1);
}

// Commenting out code for debugging
// let debugValue = computeExpensiveOperation();
let result = simpleOperation();
```

Block Comments (`/* */`):

Block comments are enclosed between `/*` and `*/` delimiters and can span multiple lines. They support nesting, making them excellent for temporarily disabling code sections and creating detailed documentation blocks.

Syntax: `/* comment content */`

Characteristics:

- Begin with `/*` (forward slash followed by asterisk)
- End with `*/` (asterisk followed by forward slash)
- Can span multiple lines
- Can appear anywhere whitespace is allowed
- Support nesting (inner `/* */` pairs are properly handled)
- Useful for large documentation blocks and code disabling

Nesting Rules:

- Each `/*` must be matched with a corresponding `*/`
- Nesting depth is tracked during lexical analysis
- Inner comments are properly nested: `/* outer /* inner */ still outer */`
- Mismatched delimiters result in lexical errors

Examples:

```
/*
 * Multi-line block comment
 * describing the following function
 * Author: John Doe
 * Date: June 2025
 */
func complexCalculation(data: [int; 10]) -> float {
    /* This algorithm implements the
       advanced mathematical formula
       discovered by Smith et al. */

    let result = 0.0;
    /* Loop through all elements */ for (item in data) {
        result = result + float(item);
    }
    return result / 10.0;  /* Calculate average */
}

/* Nested comments example */
/*
 * Outer comment begins here
 * /* Inner comment can contain code:
 *    let x = 42;
 *    let y = x * 2;
 * */
 * Outer comment continues after inner ends
 */

/* Temporarily disable entire function
func debugFunction() -> void {
    print("Debug output");
    /* Even nested comments work here
       let temp = calculate();
    */
}
*/
```

Tokens

Identifiers

Identifiers must start with a letter (a-z, A-Z) or underscore (_), followed by any combination of letters, digits (0-9), or underscores. HLang is case-sensitive.

Valid examples: `hello`, `_value`, `myVar123`

Keywords

The following are reserved keywords in HLang:

`bool`	`break`	`const`	`continue`	`else`	`false`
`float`	`for`	`func`	`if`	`in`	`int`
`let`	`return`	`string`	`true`	`void`	`while`

Operators

Arithmetic: `+`, `-`, `*`, `/`, `%`

Comparison: `==`, `!=`, `<`, `<=`, `>`, `>=`

Logical: `&&`, `||`, `!`

Assignment: `=`

Type annotation: `:`

Function return type: `->`

Pipeline: `>>`

Separators

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

Literals

Integer literals: All integers in HLang are considered as 32-bit signed integers. An integer literal consists of a sequence of one or more digits (0-9), optionally preceded by a minus sign (-) for negative values. Leading zeros are allowed but not required.

```
Examples: 42, -17, 0, 007, -0
```

Float literals: All numbers with decimal points in HLang are considered as 64-bit floating-point numbers, just as double precision floating-point numbers in C/C++. A HLang float literal consists of an integer part, a decimal part, and an optional exponent part. The integer part is a sequence of one or more digits. The decimal part is a decimal point (.) followed by zero or more digits. The exponent part starts with the character 'e' or 'E' followed by an optional '+' or '-' sign, and then one or more digits. The decimal part is

mandatory for float literals, but the fractional digits after the decimal point can be omitted. The exponent part can be omitted. An optional minus sign (-) can precede the entire literal for negative values.

```
Examples: 3.14, -2.5, 0.0, 42., 5., 1.23e10, -4.56E-3, 0.0e0
```

Boolean literals: Boolean values in HLang are represented by exactly two keywords: `true` and `false`. These are case-sensitive and cannot be abbreviated or modified.

```
Examples: true, false
```

String literals: String literals in HLang are sequences of ASCII characters enclosed in double quotes ("). A string literal can contain any ASCII character (codes 0-127) except for unescaped double quotes and backslashes, which must be represented using escape sequences. The string content between the quotes can be empty, forming an empty string. Non-ASCII characters are not supported and will result in lexical errors.

Lexical Token Representation: When tokenized, string literals return only the content between the quotes, without the enclosing double quotes. Escape sequences within the string content are preserved as-is in the token representation.

```
Examples:
- "Hello World" → token: Hello World
- "Line 1\nLine 2" → token: Line 1\nLine 2
- "Quote: \"text\"" → token: Quote: \"text\"
- "" → token: (empty string)
- "ASCII symbols: !@#$%^&*()" → token: ASCII symbols: !@#$%^&*()
```

Supported escape sequences:

Sequence	ASCII Code	Description
`\n`	10	newline character (LF)
`\t`	9	horizontal tab character
`\r`	13	carriage return character
`\\`	92	backslash character
`\"`	34	double quote character

String Content Restrictions:

- Only ASCII characters (codes 0-127) are allowed
- Non-printable ASCII characters (codes 0-31, 127) must use escape sequences where available
- Extended ASCII (codes 128-255) and Unicode characters are not supported
- Invalid characters result in lexical errors during compilation

- **Tokenization:** The lexer returns only the string content without the enclosing quotes

Array literals: Array literals in HLang are homogeneous collections represented as comma-separated values enclosed in square brackets ([ ]). The elements within an array literal must be of the same type, and the array can contain zero or more elements. When the array is empty, the type must be inferrable from context or explicitly specified.

```
Examples: [1, 2, 3], ["hello", "world"], [true, false, true], []
```

Type System

HLang uses static typing with limited type inference for local variables.

Primitive Types

Type	Description
`int`	32-bit signed integers
`float`	64-bit floating-point numbers
`bool`	Boolean values (`true` or `false`)
`string`	Immutable UTF-8 strings
`void`	Unit type for functions that don't return a value

Composite Types

Array: Homogeneous collections with fixed size determined at declaration

```
let numbers: [int; 5] = [1, 2, 3, 4, 5];        // Explicit size and type
let names = ["Alice", "Bob", "Charlie"];         // Size inferred as 3
let empty: [int; 0] = [];                        // Empty array with
explicit type
```

Function types: Represent function signatures using the pattern `(param_types...) -> return_type`

```
func add(a: int, b: int) -> int { return a + b; }
// Function type: (int, int) -> int

func greet(name: string) -> void { print("Hi " + name); }
// Function type: (string) -> void
```

Type Inference

Local variables can omit type annotations when the type can be inferred from the initializer:

```
let x = 42;                    // Inferred as int
let y = 3.14;                  // Inferred as float
let z = true;                  // Inferred as bool
let s = "hello";               // Inferred as string (literal tokenizes
as: hello)
let arr = [1, 2, 3];           // Inferred as [int; 3]
```

Variables and Constants

Variable Declarations

Variable declarations in HLang create mutable bindings using the `let` keyword. A variable declaration consists of the `let` keyword, followed by an identifier (the variable name), an optional type annotation, and an initializer expression. The general syntax is:

```
let <identifier> [: <type>] = <expression>;
```

Syntax Rules:

- The `let` keyword must be followed by a valid identifier
- Type annotations are introduced by a colon (😃 and are optional when the type can be inferred
- The assignment operator (=) separates the declaration from the initializer
- Every variable declaration must include an initializer expression
- The declaration must be terminated with a semicolon (😉

Type Inference: When a type annotation is omitted, the variable's type is inferred from the initializer expression. The type inference follows these rules:

- Integer literals infer to `int` type
- Float literals infer to `float` type
- Boolean literals infer to `bool` type
- String literals infer to `string` type
- Array literals infer array types based on element types

Mutability: All variables declared with `let` are mutable by default, meaning their values can be reassigned after declaration using the assignment operator (=).

Examples:

```
let age: int = 25;                    // Explicit type annotation
let name = "Alice";                   // Type inferred as string
let pi = 3.14159;                     // Type inferred as float
let isValid = true;                   // Type inferred as bool
```

```
let numbers = [1, 2, 3, 4, 5];        // Type inferred as [int; 5]

// Reassignment (variables are mutable)
x = 15;                               // OK, x is mutable
y = y + 5;                            // OK, arithmetic reassignment
```

Constant Declarations

Constant declarations in HLang create immutable bindings using the `const` keyword. A constant declaration consists of the `const` keyword, followed by an identifier (the constant name), an optional type annotation, and a mandatory initializer expression. The general syntax is:

```
const <identifier> [: <type>] = <expression>;
```

Syntax Rules:

- The `const` keyword must be followed by a valid identifier
- Type annotations are introduced by a colon (😃 and are optional when the type can be inferred
- The assignment operator (=) separates the declaration from the initializer
- Every constant declaration must include an initializer expression at declaration time
- The declaration must be terminated with a semicolon (😉
- Constant identifiers should follow naming conventions (typically UPPER_CASE or camelCase)

Initialization Requirements:

- Constants must be initialized at the point of declaration
- The initializer expression is evaluated at compile time for global constants
- The initializer expression is evaluated at runtime for local constants
- Once initialized, constants cannot be reassigned or modified

Type Inference: Similar to variables, when a type annotation is omitted, the constant's type is inferred from the initializer expression following the same inference rules as variables.

Immutability: All constants declared with `const` are immutable, meaning their values cannot be changed after initialization. Any attempt to reassign a constant results in a compile-time error.

Examples:

```
const MAX_SIZE: int = 100;            // Explicit type annotation
const PI = 3.14159;                   // Type inferred as float
const APP_NAME: string = "HLang";     // Explicit string type
const IS_DEBUG = false;               // Type inferred as bool
const PRIMES = [2, 3, 5, 7, 11];      // Type inferred as [int; 5]

// Error cases:
// const UNINITIALIZED: int;          // Error: missing initializer
// MAX_SIZE = 200;                    // Error: cannot reassign constant
```

Scope Rules and Visibility

Global Scope:

- Global constants declared at the program level have program-wide scope
- Global constants are visible to all functions and can be accessed throughout the program
- Global constants must be declared before any function declarations

Function Scope:

- Function parameters have function-wide scope and are visible throughout the function body
- Function parameters are immutable bindings (similar to constants)

Block Scope:

- Local variables and constants declared within blocks (enclosed by `{` and `}`) have block scope
- Block-scoped declarations are only visible within their containing block and nested blocks
- This includes variables/constants declared in function bodies, if statements, loop bodies, etc.

Shadowing Rules:

- Inner declarations can shadow (hide) outer declarations with the same name
- Shadowing applies to both variables and constants
- The innermost declaration takes precedence in case of name conflicts
- Shadowed names become inaccessible until the inner scope ends

Scope Examples:

```
const GLOBAL_CONST: int = 42;  // Global scope

func example() -> void {
    let x = 10;                // Function scope
    const LOCAL_CONST = 20;    // Function scope

    if (x > 5) {
        let y = 30;            // Block scope (if-block)
        let x = 40;            // Shadows function-scope x
        // Here: x = 40, y = 30, LOCAL_CONST = 20, GLOBAL_CONST = 42
    }
    // Here: x = 10 (original), y is not accessible
}
```

Expressions

Expressions in HLang are constructs that evaluate to a value and have a specific type. Every expression produces a result that can be used as an operand in larger expressions or assigned to variables. HLang supports several categories of expressions with well-defined evaluation rules and type semantics.

Primary Expressions

Primary expressions are the basic building blocks of all expressions in HLang:

Literal Expressions: Direct values embedded in the source code

```
42          // Integer literal
3.14        // Float literal
true        // Boolean literal
"hello"     // String literal (tokenizes as: hello)
[1, 2, 3]   // Array literal
```

Identifier Expressions: References to variables, constants, or functions

```
let x = 10;
const PI = 3.14159;
func add(a: int, b: int) -> int { return a + b; }

// Identifier expressions:
x           // Variable reference
PI          // Constant reference
add         // Function reference
```

Parenthesized Expressions: Expressions enclosed in parentheses for grouping

```
(x + y) * z     // Parentheses change evaluation order
(condition)     // Explicit grouping for clarity
```

Arithmetic Expressions

Arithmetic expressions perform mathematical computations on numeric operands. HLang supports both binary and unary arithmetic operators with standard mathematical semantics.

Binary Arithmetic Operators:

Operator	Description	Operand Types	Result Type	Notes
+	Addition	int, float	int/float	Also string concatenation
-	Subtraction	int, float	int/float	
*	Multiplication	int, float	int/float	
/	Division	int, float	int/float	Division by zero is runtime error
%	Modulo	int only	int	Remainder after division

Unary Arithmetic Operators:

Operator	Description	Operand Types	Result Type	Notes
`-`	Negation	int, float	int/float	Unary minus (prefix)
`+`	Unary plus	int, float	int/float	Identity operation (optional)

Type Coercion and Promotion Rules:

- `int` operands with `int` operands result in `int` type
- `float` operands with `float` operands result in `float` type
- Mixed `int` and `float` operands: `int` is promoted to `float`, result is `float`
- String concatenation: `string + any_type` converts the non-string operand to string representation

Division Semantics:

- `int / int`: Performs integer division (truncates toward zero)
- `float / float`, `int / float`, `float / int`: Performs floating-point division

Examples:

```
let a = 10 + 5;          // 15 (int + int = int)
let b = 3.5 + 2;         // 5.5 (float + int = float)
let c = 10 / 3;          // 3 (integer division)
let d = 10.0 / 3;        // 3.333... (float division)
let e = 15 % 4;          // 3 (modulo)
let f = -x;              // Negation
let g = "Count: " + 42; // "Count: 42" (string concatenation)
```

Comparison Expressions

Comparison expressions evaluate the relationship between two operands and always produce a boolean result. All comparison operators have the same precedence and are left-associative.

Equality Operators:

Operator	Description	Operand Types	Result Type
`==`	Equality	int, float, bool, string	bool
`!=`	Inequality	int, float, bool, string	bool

Relational Operators:

Operator	Description	Operand Types	Result Type
`<`	Less than	int, float, string	bool
`<=`	Less than or equal	int, float, string	bool
`>`	Greater than	int, float, string	bool

Operator	Description	Operand Types	Result Type
`>=`	Greater than or equal	int, float, string	bool

Comparison Semantics:

- **Numeric comparisons:** Follow standard mathematical ordering
- **String comparisons:** Use lexicographic (dictionary) ordering based on Unicode code points
- **Boolean comparisons:** `false < true` (false is considered less than true)
- **Mixed type comparisons:** `int` and `float` can be compared (with type promotion)
- **Array comparisons:** Not supported (compile-time error)

Examples:

```
let eq1 = (5 == 5);           // true
let eq2 = (3.14 == 3.14);     // true
let ne1 = (10 != 5);          // true
let lt1 = (3 < 5);            // true
let lt2 = (2.5 < 3);          // true (float < int)
let str1 = ("abc" < "def");   // true (lexicographic)
let str2 = ("hello" == "hello"); // true
```

Logical Expressions

Logical expressions operate on boolean operands and implement boolean algebra operations. HLang supports three logical operators with short-circuit evaluation semantics.

Logical Operators:

Operator	Description	Operand Types	Result Type	Evaluation
`!`	Logical NOT	bool	bool	Unary prefix
`&&`	Logical AND	bool	bool	Short-circuit
`||`	Logical OR	bool	bool	Short-circuit

Short-Circuit Evaluation:

- **AND (`&&`):** If left operand is `false`, right operand is not evaluated (result is `false`)
- **OR (`||`):** If left operand is `true`, right operand is not evaluated (result is `true`)
- This behavior is important for expressions with side effects

Truth Tables:

NOT Operator (`!`):

Input	Output
`true`	`false`

Input	Output
false	true

AND Operator (&&):

Left	Right	Result
true	true	true
true	false	false
false	true	false
false	false	false

OR Operator (||):

Left	Right	Result
true	true	true
true	false	true
false	true	true
false	false	false

Examples:

```
let not1 = !true;                    // false
let not2 = !false;                   // true
let and1 = true && false;            // false
let and2 = (x > 0) && (y < 10);     // Conditional AND
let or1 = true || false;             // true
let or2 = (x == 0) || (y == 0);     // Conditional OR

// Short-circuit examples:
let safe = (arr != null) && (arr[0] > 5);  // Won't access arr[0] if arr
is null
let found = (result != null) || search();  // Won't call search() if
result exists
```

Array Access Expressions

Array access expressions retrieve or modify elements of array values using index notation. Arrays in HLang use zero-based indexing.

Syntax: `array_expression[index_expression]`

Semantics:

- The array expression must evaluate to an array type

- The index expression must evaluate to an `int` type
- Index must be within bounds: `0 <= index < array_length`
- Out-of-bounds access results in a runtime error
- Array access can be used for both reading and writing (if array is mutable)

Examples:

```
let numbers: [int; 5] = [10, 20, 30, 40, 50];
let first = numbers[0];        // 10 (read access)
let last = numbers[4];         // 50 (read access)
numbers[2] = 35;              // Modify element (write access)

// Dynamic indexing:
let index = 1;
let value = numbers[index];    // 20
```

Multi-Dimensional Arrays:

HLang supports multi-dimensional arrays through arrays of arrays. Each dimension is accessed using separate index expressions applied sequentially.

Syntax: `array_expression[index1][index2]...[indexN]`

Declaration and Initialization:

```
// 2D array (matrix): 2 rows, 3 columns each
let matrix: [[int; 3]; 2] = [[1, 2, 3], [4, 5, 6]];

// 3D array: 2 layers, 3 rows, 4 columns each
let cube: [[[int; 4]; 3]; 2] = [
    [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]],
    [[13, 14, 15, 16], [17, 18, 19, 20], [21, 22, 23, 24]]
];

// Jagged arrays (arrays of different-sized arrays)
let jagged: [[int; 2]; 3] = [[1, 2], [3, 4], [5, 6]];
```

Multi-Dimensional Access:

```
// 2D array access:
let element = matrix[1][2];    // 6 (row 1, column 2)
matrix[0][1] = 99;            // Modify element in row 0, column 1

// 3D array access:
let value = cube[1][2][3];     // 24 (layer 1, row 2, column 3)
cube[0][1][2] = 100;          // Modify element

// Sequential access (equivalent to above):
```

```
let row = matrix[1];           // Get entire row: [4, 5, 6]
let element2 = row[2];         // 6 (same as matrix[1][2])
```

Bounds Checking:

- Each dimension is bounds-checked independently
- Out-of-bounds access in any dimension causes a runtime error
- Example: `matrix[2][1]` is invalid if matrix has only 2 rows (indices 0, 1)

Type Semantics:

```
// Type breakdown for [[int; 3]; 2]:
// - Outer array: [T; 2] where T = [int; 3]
// - Inner arrays: [int; 3]
// - Elements: int

let matrix: [[int; 3]; 2] = [[1, 2, 3], [4, 5, 6]];
// matrix has type [[int; 3]; 2]
// matrix[0] has type [int; 3]
// matrix[0][1] has type int
```

Function Call Expressions

Function call expressions invoke functions with specified arguments and evaluate to the function's return value.

Syntax: `function_expression(argument_list)`

Argument List: Zero or more expressions separated by commas, enclosed in parentheses

Semantics:

- Function expression must evaluate to a function type
- Number of arguments must match function's parameter count
- Argument types must be compatible with parameter types
- Arguments are evaluated left-to-right before function invocation
- Function body is executed with arguments bound to parameters
- Expression evaluates to the function's return value

Examples:

```
// Simple function calls:
let sum = add(5, 3);                    // Call with literal arguments
let result = multiply(x, y);            // Call with variable arguments
print("Hello, World!");                 // Void function call

// Nested function calls:
let nested = add(multiply(2, 3), 4);    // 10 (6 + 4)
let chain = sqrt(abs(-16));             // Function composition
```

```
// Function expressions as arguments:
let mapped = map(numbers, square);       // Higher-order function
```

Operator Precedence and Associativity

HLang defines operator precedence to determine evaluation order in complex expressions. Higher precedence operators are evaluated before lower precedence ones. The pipeline operator (`>>`) is integrated into this precedence system.

Complete Precedence Table (Highest to Lowest):

Level	Operators	Description	Associativity
1	`() []`	Function call, Array access	Left
2	`! - +`	Unary NOT, negation, plus	Right
3	`* / %`	Multiplicative operators	Left
4	`+ -`	Additive operators	Left
5	`< <= > >=`	Relational operators	Left
6	`== !=`	Equality operators	Left
7	`&&`	Logical AND	Left
8	`||`	Logical OR	Left
9	`>>`	Pipeline operator	Left

Associativity Rules:

- **Left associative:** `a op b op c` evaluates as `(a op b) op c`
- **Right associative:** `a op b op c` evaluates as `a op (b op c)`

Precedence Examples:

```
let expr1 = 2 + 3 * 4;              // 14 (not 20) - multiplication first
let expr2 = (2 + 3) * 4;            // 20 - parentheses override
precedence
let expr3 = x < 5 && y > 10;        // (x < 5) && (y > 10) - comparison
first
let expr4 = !flag && condition;     // (!flag) && condition - unary first
let expr5 = a + b < c + d;          // (a + b) < (c + d) - addition before
comparison
let expr6 = 5 + 3 >> multiply(2);   // (5 + 3) >> multiply(2) = 16 -
pipeline lowest
let expr7 = data >> filter(isValid) >> map(transform); // Left-associative
chaining
```

Type Checking and Evaluation

Static Type Checking: All expressions are type-checked at compile time to ensure type safety

Evaluation Order: Expressions are evaluated according to operator precedence and associativity rules, with subexpressions evaluated as needed

Side Effects: Most expressions in HLang are pure (no side effects), except for function calls which may have side effects

---

Execution Pipeline Syntax

HLang addresses the traditional limitations of nested function calls and complex expression composition through its distinctive Pipeline Operator (`>>`). This syntactic innovation allows data to flow linearly through a series of transformations, making code more readable and intuitive while maintaining the functional programming paradigm's benefits.

Pipeline Operator (`>>`)

The pipeline operator passes the result of the left expression as the first argument to the right expression. This creates a left-to-right data flow that mirrors how programmers naturally think about sequential transformations.

Syntax: `expression >> function_call`

Semantics:

- The left operand is evaluated first
- The result becomes the first argument to the function on the right
- If the right operand expects multiple parameters, additional arguments follow normally
- The pipeline operator has left associativity, enabling chaining
- Type checking ensures the left operand type matches the first parameter type of the right function

Basic Pipeline Examples:

```
// Traditional nested calls (hard to read):
let result1 = addExclamation(uppercase(trim(" hello world ")));

// Pipeline equivalent (readable left-to-right):
let result2 = " hello world " >> trim >> uppercase >> addExclamation;

// Multi-parameter function in pipeline:
let numbers = [1, 2, 3, 4, 5];
let doubled = numbers >> map(multiply_by_two) >> filter(is_even);

// Mathematical operations in pipeline:
let calculation = 5 >> add(3) >> multiply(2) >> subtract(4);  // ((5 + 3)
* 2) - 4 = 12
```

Pipeline with Mixed Parameter Functions:

```
// Function with multiple parameters:
func formatMessage(text: string, prefix: string, suffix: string) -> string
{
    return prefix + text + suffix;
}

// Pipeline passes first argument, others provided normally:
let formatted = "Hello" >> formatMessage("[", "]");  // "[Hello]"

// Equivalent to:
let formatted2 = formatMessage("Hello", "[", "]");
```

Pipeline Type Checking:

```
func processInt(x: int) -> string { return "Number: " + str(x); }
func processString(s: string) -> string { return s + "!"; }

// Valid pipeline (types match):
let result = 42 >> processInt >> processString;  // "Number: 42!"

// Invalid pipeline (type mismatch):
// let invalid = "hello" >> processInt;  // Error: string cannot be passed
to processInt
```

Pipeline Operator Precedence and Examples

The pipeline operator has the lowest precedence among binary operators (level 9), ensuring that arithmetic and logical operations are evaluated before pipeline transformation.

Precedence Examples with Pipeline:

```
// Pipeline has lower precedence than arithmetic:
let result = 5 + 3 >> multiply(2);     // (5 + 3) >> multiply(2) = 16

// Parentheses override precedence:
let result2 = 5 + (3 >> multiply(2));  // 5 + (3 * 2) = 11

// Multiple pipelines (left-associative):
let chain = data >> filter(isValid) >> map(transform) >> reduce(combine);
// Equivalent to: ((data >> filter(isValid)) >> map(transform)) >>
reduce(combine)

// Mixed operators:
let complex = (x > 0) && (data >> process >> validate);
// Comparison and logical operations before pipeline
```

Pipeline Performance Considerations

Optimization Characteristics:

- Pipeline operations are optimized at compile time where possible
- Intermediate results may be eliminated through fusion optimization
- Memory allocation is minimized through lazy evaluation strategies
- Type checking occurs at compile time, not runtime

Performance Examples:

```
// Efficient pipeline (fused operations):
let optimized = data >> map(transform) >> filter(isValid) >> take(10);
// Compiler may fuse these operations into a single pass

// Memory-efficient processing:
let stream = largeDataset >>
    filter(criteria) >>
    map(expensive_transform) >>
    take(5);  // Only processes first 5 valid items
```

Pipeline Integration with Type System

Pipeline operations are fully integrated with HLang's static type system, ensuring type safety throughout the transformation chain.

Type Inference in Pipelines:

```
// Type inference through pipeline:
let result = "123" >> int >> add(5) >> str;
// Types: string -> int -> int -> string

// Generic pipeline functions:
func pipeline<T, U, V>(input: T, f1: T -> U, f2: U -> V) -> V {
    return input >> f1 >> f2;
}

// Array pipeline with type preservation:
let numbers = [1, 2, 3, 4, 5];
let processed: [string; 5] = numbers >> map(str) >> sort;
```

Pipeline Function Signatures:

```
// Pipeline-aware function definitions:
func pipelineProcessor(input: string) -> string {
    // This function is designed for pipeline use
    return input >> validate >> clean >> format;
}
```

```
// Higher-order pipeline functions:
func createPipeline(processor: string -> string) -> (string -> string) {
    return func(input: string) -> string {
        return input >> processor >> finalize;
    };
}
```

Statements

Statements in HLang are executable constructs that perform actions but do not produce values (unlike expressions). Statements form the imperative backbone of HLang programs and control program execution flow. Every statement must be terminated with a semicolon (`;`) except for compound statements that use block syntax.

Expression Statements

Expression statements execute expressions for their side effects and discard the resulting value. They consist of any valid expression followed by a semicolon.

Syntax: `expression;`

Purpose:

- Execute functions for their side effects (e.g., I/O operations)
- Perform assignments (though assignment is technically an expression)
- Evaluate expressions that modify program state

Examples:

```
x = 10;                    // Assignment expression as statement
print("Hello World");      // Function call for side effect
factorial(5);              // Function call (result discarded)
arr[0] = 42;              // Array element assignment
++counter;                 // Increment operation (if supported)
```

Semantics:

- The expression is fully evaluated
- Any side effects of the expression occur
- The resulting value is discarded
- Execution continues to the next statement

Variable Declaration Statements

Variable declaration statements introduce new variables into the current scope with initialization. These statements create mutable bindings that can be reassigned later.

Syntax: `let identifier [: type] = expression;`

Components:

- **`let` keyword:** Indicates variable declaration
- **Identifier:** Variable name (must be valid identifier)
- **Type annotation:** Optional explicit type specification
- **Initializer:** Mandatory expression providing initial value

Type Inference Rules:

- If type annotation is present, the initializer must be compatible with the specified type
- If type annotation is omitted, the variable type is inferred from the initializer expression
- Type inference occurs at compile time

Scope and Lifetime:

- Variables are scoped to the block in which they are declared
- Variables exist from declaration point until end of containing block
- Variables can shadow outer-scope variables with the same name

Examples:

```
let age: int = 25;                    // Explicit type annotation
let name = "Alice";                   // Type inferred as string
let pi = 3.14159;                     // Type inferred as float
let numbers = [1, 2, 3, 4, 5];       // Type inferred as [int; 5]
let isValid: bool = checkInput();     // Type annotation with function
call

// Shadowing example:
let x = 10;                           // Outer x (int)
{
    let x = "hello";                  // Inner x (string) - shadows outer
x
    // Inner x is accessible here
}
// Outer x is accessible again here
```

Assignment Statements

Assignment statements modify the value of existing variables or array elements. The target of assignment must be a mutable lvalue (assignable location).

Syntax: `lvalue = expression;`

Valid Lvalues:

- Variable identifiers (declared with `let`)
- Array element access expressions (`array[index]`)
- Multi-dimensional array access (`matrix[i][j]`)

Semantics:

- Right-hand side expression is evaluated first
- Result type must be compatible with lvalue type
- Value is assigned to the lvalue location
- Assignment expression evaluates to the assigned value

Type Checking:

- Assignment target must be mutable (not `const`)
- Expression type must match or be convertible to target type
- Array bounds are checked at runtime for array element assignments

Examples:

```
// Variable assignment:
let x: int = 10;
x = 20;                              // Simple assignment
x = x + 5;                           // Self-referential assignment

// Array element assignment:
let numbers: [int; 3] = [1, 2, 3];
numbers[0] = 10;                     // Single element
numbers[1] = numbers[2] + 5;         // Expression assignment

// Multi-dimensional assignment:
let matrix: [[int; 2]; 2] = [[1, 2], [3, 4]];
matrix[0][1] = 99;                   // 2D array assignment

// Error cases:
const PI = 3.14159;
// PI = 2.71828;                     // Error: cannot assign to const
// numbers[5] = 10;                  // Runtime error: index out of bounds
```

Conditional Statements

Conditional statements execute different code paths based on boolean expressions. HLang supports `if`, `else if`, and `else` constructs for conditional execution.

Syntax:

```
if (condition_expression) {
    statements
} [else if (condition_expression) {
    statements
}]* [else {
    statements
}]?
```

Condition Requirements:

- Condition expressions must evaluate to `bool` type
- No implicit type conversion to boolean (unlike C/C++)
- Parentheses around condition are mandatory

Execution Semantics:

- Conditions are evaluated in order (top to bottom)
- First `true` condition executes its corresponding block
- Only one block is executed per conditional statement
- If no condition is `true` and `else` exists, `else` block executes
- If no condition is `true` and no `else` exists, execution continues after the conditional

Examples:

```
// Simple if statement:
if (age >= 18) {
    print("Adult");
}

// If-else statement:
if (score >= 90) {
    grade = "A";
} else {
    grade = "B";
}

// Multiple conditions:
if (temperature > 30) {
    print("Hot weather");
} else if (temperature > 20) {
    print("Warm weather");
} else if (temperature > 10) {
    print("Cool weather");
} else {
    print("Cold weather");
}

// Nested conditionals:
if (user.isLoggedIn) {
    if (user.isAdmin) {
        showAdminPanel();
    } else {
        showUserPanel();
    }
}

// Complex conditions:
if ((age >= 18 && hasLicense) || isEmergency) {
    allowDriving = true;
}
```

Loop Statements

Loop statements enable repetitive execution of code blocks. HLang supports two types of loops: `while` loops for condition-based iteration and `for` loops for collection-based iteration.

While Loops

While loops repeatedly execute a block of statements as long as a condition remains true.

Syntax:

```
while (condition_expression) {
    statements
}
```

Execution Semantics:

- Condition is evaluated before each iteration (pre-test loop)
- If condition is `true`, loop body executes
- After loop body, condition is re-evaluated
- Loop terminates when condition becomes `false`
- If condition is initially `false`, loop body never executes

Examples:

```
// Basic counting loop:
let i = 0;
while (i < 10) {
    print("Count: " + str(i));
    i = i + 1;
}

// Input validation loop:
let input: string;
while (input != "quit") {
    input = getUserInput();
    processInput(input);
}

// Infinite loop (requires break to exit):
while (true) {
    let command = getCommand();
    if (command == "exit") {
        break;
    }
    executeCommand(command);
}
```

For Loops (Iterator-based)

For loops iterate over collections, automatically handling iteration logic and providing access to each element.

Syntax:

```
for (variable_name in collection_expression) {
    statements
}
```

Semantics:

- Collection expression must evaluate to an array type
- Loop variable is automatically declared with element type
- Loop variable is scoped to the loop body
- Each iteration assigns the next element to the loop variable
- Loop terminates when all elements have been processed

Examples:

```
// Array iteration:
let numbers = [1, 2, 3, 4, 5];
for (num in numbers) {
    print("Number: " + str(num));
}

// String iteration (if strings are iterable):
let text = "Hello";
for (char in text) {
    print("Character: " + char);
}

// Multi-dimensional array iteration:
let matrix = [[1, 2], [3, 4], [5, 6]];
for (row in matrix) {
    for (element in row) {
        print(str(element));
    }
}

// Processing with conditions:
let scores = [85, 92, 78, 96, 88];
for (score in scores) {
    if (score >= 90) {
        print("Excellent: " + str(score));
    }
}
```

Control Flow Statements

Control flow statements alter the normal sequential execution of statements within loops and functions.

Break Statement

The `break` statement immediately terminates the nearest enclosing loop and transfers control to the statement following the loop.

Syntax: `break;`

Semantics:

- Only valid within loop bodies (`while` or `for` loops)
- Terminates the innermost containing loop
- Execution resumes after the loop's closing brace
- Compile-time error if used outside a loop

Examples:

```
// Early loop termination:
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
for (num in numbers) {
    if (num > 5) {
        break;  // Exit loop when number exceeds 5
    }
    print(str(num));
}
// Prints: 1, 2, 3, 4, 5

// Search with early exit:
let found = false;
let target = 7;
for (value in data) {
    if (value == target) {
        found = true;
        break;  // Stop searching once found
    }
}
```

Continue Statement

The `continue` statement skips the remaining statements in the current loop iteration and proceeds to the next iteration.

Syntax: `continue;`

Semantics:

- Only valid within loop bodies

- Skips remaining statements in current iteration
- For `while` loops: jumps to condition evaluation
- For `for` loops: proceeds to next collection element
- Compile-time error if used outside a loop

Examples:

```
// Skip even numbers:
for (i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) {
    if (i % 2 == 0) {
        continue;  // Skip even numbers
    }
    print("Odd: " + str(i));
}
// Prints: Odd: 1, Odd: 3, Odd: 5, Odd: 7, Odd: 9

// Data filtering:
for (item in dataList) {
    if (!isValid(item)) {
        continue;  // Skip invalid items
    }
    processItem(item);
}
```

Return Statement

The `return` statement terminates function execution and optionally returns a value to the caller.

Syntax:

- `return;` (for void functions)
- `return expression;` (for non-void functions)

Semantics:

- Immediately terminates current function execution
- For non-void functions: expression type must match declared return type
- For void functions: no expression allowed
- Control returns to the function call site
- Functions must have return statements on all execution paths (except void functions)

Examples:

```
// Void function return:
func printGreeting(name: string) -> void {
    if (name == "") {
        return;  // Early return for empty name
    }
    print("Hello, " + name + "!");
```

```
}

// Non-void function return:
func factorial(n: int) -> int {
    if (n <= 1) {
        return 1;  // Base case
    }
    return n * factorial(n - 1);  // Recursive case
}

// Multiple return paths:
func findMax(a: int, b: int) -> int {
    if (a > b) {
        return a;
    } else {
        return b;
    }
    // All paths must return a value
}

// Early return pattern:
func processUser(user: User) -> bool {
    if (!user.isValid()) {
        return false;  // Early failure return
    }

    if (!user.hasPermission()) {
        return false;  // Another early return
    }

    // Main processing logic
    user.process();
    return true;  // Success return
}
```

Block Statements

Block statements group multiple statements together and create new lexical scopes. Blocks are delimited by curly braces `{` and `}`.

Syntax:

```
{
    statements
}
```

Scope Rules:

- Variables declared within a block are scoped to that block
- Inner blocks can access variables from outer blocks

- Variables declared in inner blocks can shadow outer variables
- Block scope ends at the closing brace

Examples:

```
// Explicit block for scoping:
{
    let tempVar = computeValue();
    let result = processValue(tempVar);
    // tempVar and result are not accessible outside this block
}

// Nested scoping:
let x = 10;
{
    let y = 20;
    {
        let z = 30;
        // x, y, z all accessible here
        let x = 100;  // Shadows outer x
        // Here: x = 100, y = 20, z = 30
    }
    // Here: x = 10 (original), y = 20, z not accessible
}
// Here: only x = 10 accessible
```

Functions

Functions in HLang are first-class constructs that encapsulate reusable code blocks with well-defined interfaces. They support parameters, return values, and local scope isolation. Functions are the primary mechanism for code organization and modularity in HLang programs.

Function Declarations

Function declarations define named functions with specified parameter lists and return types. Every function declaration establishes a new scope and defines a callable entity.

Syntax:

```
func function_name(parameter_list) -> return_type {
    function_body
}
```

Components:

- `func` **keyword:** Introduces function declaration
- **Function name:** Valid identifier following naming conventions

- **Parameter list:** Zero or more parameter declarations separated by commas
- **Return type:** Type of value returned by function (or `void` for no return)
- **Function body:** Block statement containing function implementation

Parameter Declaration Syntax:

```
parameter_name: parameter_type
```

Complete Examples:

```
// Function with multiple parameters:
func add(a: int, b: int) -> int {
    return a + b;
}

// Function with no parameters:
func getCurrentTime() -> string {
    return getSystemTime();
}

// Void function:
func printGreeting(name: string) -> void {
    print("Hello, " + name + "!");
}

// Function with array parameter:
func sumArray(numbers: [int; 5]) -> int {
    let total = 0;
    for (num in numbers) {
        total = total + num;
    }
    return total;
}

// Function with mixed parameter types:
func formatScore(name: string, score: int, isPassing: bool) -> string {
    let status = isPassing ? "PASS" : "FAIL";
    return name + ": " + str(score) + " (" + status + ")";
}
```

Function Signatures and Type System

Function Types: Every function has a type that describes its parameter types and return type. Function types follow the pattern `(param_types...) -> return_type`.

```
func multiply(x: int, y: int) -> int { return x * y; }
// Type: (int, int) -> int
```

```
func greet(name: string) -> void { print("Hi " + name); }
// Type: (string) -> void

func getPI() -> float { return 3.14159; }
// Type: () -> float
```

Function Type Compatibility:

- Functions are compatible if parameter types and return types match exactly
- Parameter names do not affect type compatibility
- Parameter order must match exactly

Parameter Passing Semantics

HLang uses different parameter passing mechanisms depending on the data type, balancing performance with safety and predictability.

Pass-by-Value (Primitive Types):

- Applied to: `int`, `float`, `bool`
- Behavior: Copies the value to the function parameter
- Modifications inside function do not affect original variable
- Performance: Efficient for small data types

```
func modifyInt(x: int) -> void {
    x = 100;  // Only modifies local parameter copy
}

let value = 42;
modifyInt(value);
// value is still 42 after function call
```

Pass-by-Value-Semantics (Strings):

- Applied to: `string`
- Behavior: Strings are immutable, so effectively pass-by-value
- No copying overhead due to immutability
- Cannot modify string content within function

```
func processString(text: string) -> string {
    // Cannot modify text parameter directly
    return text + " processed";
}

let original = "data";
let result = processString(original);
// original remains "data", result is "data processed"
```

Pass-by-Reference (Arrays):

- Applied to: All array types `[T; N]`
- Behavior: Passes reference to original array
- Modifications inside function affect the original array
- Performance: Efficient for large data structures

```
func fillArray(arr: [int; 3], value: int) -> void {
    for (i in [0, 1, 2]) {
        arr[i] = value;  // Modifies original array
    }
}

let numbers = [1, 2, 3];
fillArray(numbers, 99);
// numbers is now [99, 99, 99]
```

Function Scope and Local Variables

Function Scope Rules:

- Each function creates its own lexical scope
- Function parameters are scoped to the function body
- Local variables declared within function are scoped to their containing block
- Functions can access global constants but not global variables (if any)

Variable Lifetime:

- Parameters exist for the duration of function execution
- Local variables exist from declaration until end of containing scope
- Return values are copied out before function scope ends

Scope Examples:

```
let globalConst = 100;  // Global scope

func example(param: int) -> int {
    // param is function-scoped
    let localVar = param * 2;  // Function-scoped local variable

    {
        let blockVar = localVar + 1;  // Block-scoped variable
        if (blockVar > 10) {
            let conditionVar = blockVar / 2;  // Conditional block scope
            return conditionVar;
        }
        // conditionVar not accessible here
    }
    // blockVar not accessible here
```

```
    return localVar + globalConst;  // Can access global const
}
```

Return Statements and Control Flow

Return Statement Requirements:

- Non-void functions must return a value on all execution paths
- Void functions can optionally use `return;` for early exit
- Return type must exactly match declared function return type
- Functions must have at least one return statement (except void functions with no early returns)

Control Flow Analysis:

```
// Valid: All paths return
func absoluteValue(x: int) -> int {
    if (x >= 0) {
        return x;     // Path 1: return positive value
    } else {
        return -x;    // Path 2: return negated value
    }
    // Both paths covered
}

// Valid: Single return path
func factorial(n: int) -> int {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}

// Error: Missing return on some paths
func badFunction(x: int) -> int {
    if (x > 0) {
        return x;
    }
    // Error: No return for x <= 0 case
}

// Valid: Void function with early return
func printPositive(x: int) -> void {
    if (x <= 0) {
        return;  // Early exit for non-positive values
    }
    print("Positive: " + str(x));
}
```

Function Overloading and Name Resolution

Function Names:

- Function names must be unique within their scope
- HLang does not support function overloading (multiple functions with same name)
- Function names follow identifier naming rules
- Function names are case-sensitive

Name Resolution:

- Function calls resolve to the function with matching name in scope
- Local function declarations shadow global ones (if supported)
- Function name lookup occurs at compile time

Recursion

HLang supports both direct and indirect recursion with proper tail-call handling.

Direct Recursion:

```
func factorial(n: int) -> int {
    if (n <= 1) {
        return 1;                    // Base case
    }
    return n * factorial(n - 1);     // Recursive call
}

func fibonacci(n: int) -> int {
    if (n <= 1) {
        return n;                    // Base cases: fib(0)=0, fib(1)=1
    }
    return fibonacci(n - 1) + fibonacci(n - 2);  // Recursive calls
}
```

Indirect Recursion:

```
func isEven(n: int) -> bool {
    if (n == 0) {
        return true;
    }
    return isOdd(n - 1);             // Calls isOdd
}

func isOdd(n: int) -> bool {
    if (n == 0) {
        return false;
    }
    return isEven(n - 1);            // Calls isEven
}
```

Recursion Guidelines:

- Always include base cases to prevent infinite recursion
- Ensure recursive calls progress toward base cases
- Be aware of stack depth limitations for deep recursion
- Consider iterative alternatives for performance-critical code

Built-in Functions

HLang provides a standard library of built-in functions for common operations. These functions are automatically available without import.

Input/Output Functions:

Function	Signature	Description
`print`	`(string) -> void`	Output string to standard output with newline
`input`	`() -> string`	Read line from standard input

```
print("Enter your name:");        // Output prompt
let name = input();               // Read user input
print("Hello, " + name + "!");    // Output greeting
```

Type Conversion Functions:

HLang provides overloaded conversion functions for type transformations:

Function	Input Type	Output Type	Description
`str`	`int`	`string`	Convert integer to string representation
`str`	`float`	`string`	Convert float to string representation
`str`	`bool`	`string`	Convert boolean to string ("true" or "false")
`int`	`string`	`int`	Parse string to integer (runtime error if invalid)
`float`	`string`	`float`	Parse string to float (runtime error if invalid)

```
// String conversion examples:
let age = 25;
let ageStr = str(age);            // "25"

let pi = 3.14159;
let piStr = str(pi);              // "3.14159"

let isValid = true;
let validStr = str(isValid);      // "true"

// Parsing examples:
```

```
let numStr = "42";
let num = int(numStr);            // 42

let floatStr = "3.14";
let floatNum = float(floatStr);   // 3.14

// Error cases (runtime errors):
// let invalid = int("not_a_number");  // Runtime error
// let invalid2 = float("abc");        // Runtime error
```

Array Utility Functions:

Function	Signature	Description
`len`	`([T; N]) -> int`	Get the length of an array

```
let numbers = [1, 2, 3, 4, 5];
let arrayLength = len(numbers);   // 5

let names = ["Alice", "Bob"];
let nameCount = len(names);       // 2
```

Example Programs

Factorial Calculation

```
func factorial(n: int) -> int {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}

func main() -> void {
    let num = 5;
    let result = factorial(num);
    print("Factorial of " + str(num) + " is " + str(result));
}
```

Array Processing

```
func sum_array(arr: [int; 5]) -> int {
    let total = 0;
    for (element in arr) {
        total = total + element;
```

```
    }
    return total;
}

func main() -> void {
    let numbers: [int; 5] = [1, 2, 3, 4, 5];
    let sum = sum_array(numbers);
    print("Sum of array: " + str(sum));
}
```

Simple Calculator

```
func add(a: float, b: float) -> float {
    return a + b;
}

func multiply(a: float, b: float) -> float {
    return a * b;
}

func main() -> void {
    let x = 10.5;
    let y = 3.2;

    print("Addition: " + str(add(x, y)));
    print("Multiplication: " + str(multiply(x, y)));
}
```

Conclusion

HLang provides a balanced approach to teaching compiler construction. It includes essential language features while avoiding the complexity pitfalls that can overwhelm students. The static type system with limited inference gives students experience with type checking without the full complexity of advanced type inference algorithms.

The language is designed to scale well across all four assignment phases, with each phase building naturally on the previous one. Students can focus on learning compiler techniques rather than struggling with language complexity.
