# noC Compiler Internals

## *Type Checking, First Class Functions, Virtual Threads, Garbage Collection and Non Blocking I/O*

Umar Karimabadi

Version 1.0, January 27, 2026: First Edition

# Table of Contents

# Chapter 1. Introduction

## 1.1. Preface

**umar**

This book acts as a reference for how key features in the noC Compiler work. This book will focus on describing the data structures that are needed for each phase of compilation and for the runtime of the language. More emphasis will be provided on the "description" at a high level than the actual implementation details. In my experience, describing the process with brief code snippets is enough to build an intuition; pasting details of source not only inflates the book but provides too much opinionated information. Code snippets that are shared will be greatly simplified with what is in the code base.

By itself this book will not be a good learning resource to understanding compiler construction and runtimes. This is why I have created a YouTube video :: "Understanding Compilers and Runtimes - Type Checking, Generics, Virtual Threads, Garbage Collection"

Although it is not required for the reader to have a basic understanding of compiler construction, a familiarity with the subject material before reading this document will certainly help. If you watch the YouTube video, no prior background is required as I go through the entire compiler in great depth.

My motivations for creating a language were firstly to have my own programming language that I would enjoy writing programs. I wanted an incredibly simple language with high-level features with a clear and concise syntax. Anybody should be able to read a noC program and understand what the program is trying to do.

Secondly, creating a compiler and runtime forces one to expose themselves to a significant amount of material in order to implement a language. From Type Checking to CPU cache consistency to non-blocking I/O, you end up exploring a lot of material and immediately applying it to implement the language.

Thirdly, the runtime allows us to have a sandbox where we can do a wide variety of testing that would not be afforded to us had we tried to do the same level of testing in a runtime like Java/Go. These runtimes (for good reason) are incredibly massive and it is difficult to run precise experimentation on these platforms.

The noC language runtime has a scheduler to support virtual threads and a garbage collector for automatic memory management. With the runtime in place, we can start to imagine how we might improve and iterate across these key components, and also measure the performance of the language as we add more complexity into the runtime.

The iteration and improvement become simple as each line of code was written by me. The first version released for this compiler and runtime executes code slowly as it does not compile directly to machine code but interprets a virtual instruction set. No effort has been made into profiling the runtime, but the Sandbox is there to be played with.

## 1.2. Coding Styles

**umar**

The noC programming language is created in C++; however, I have ensured that the C++ I use is incredibly simple to understand; this was mainly for me and it turned out make the code more approachable. An understanding of Java or any similar high-level object-oriented language is enough to read the source code.

Everything in the code base is public, as I am the only one developing on this code base, I purposely will not sabotage myself by breaking encapsulation layers.

More "modern" (it’s 2026 so I guess these are considered old) features like unique pointers and shared pointers were not used in the code base because I find them quite ugly to look at (and type!) and if I replaced everything with shared pointers, the codebase will have the same issues with object lifetimes.

Const referencing and the const keyword are rarely used unless forced upon me by some standard library or the compiler.

There is no consistent use of `auto` or any consistent use of casing, snake casing and camel case are both used throughout the code base.

Safe casts for signdidness comparisons are used only in the case where I know the cast will cause bugs. In some places I am fine with comparing an `int` to a `size_t` and ignore warnings Also C style casting is used instead of static_cast/reinterpret cast.

`assert` and `assert_never_reaches` are heavily used in the codebase. I find it very useful.

The only C++ feature that readers need to be familiar with is Destructors/RAII; when an object falls out of scope, its destructor function is called.

There is only one use of operator overloading in the entire source code and that is to make some bitwise operations simpler to write.

I hope those who do read the code can navigate it well enough to learn something from it.

## 1.3. Testing

**umar**

I have decided to publish this document with the language still needing more testing. Although there are significant examples in the noC_dir, there are still many bugs to be found in all phases of compilation. There are bound to be signficant bugs in the runtime and garbage collector as I did not have time to throughly test them or complete the implementation fully.

There are fundamental issues with lifetimes of objects in the compiler. Shortcuts were taken to reference objects to make code generation and runtime easier, but this ended up forcing many objects to share global lifetimes even if they are not needed.

Unfortunately, I gave myself a hard deadline to release the source code and this document. I have other commitments that I must attend to and no longer have time to work on the langauge to fix these issues. Nevertheless, I am still happy with the language and more than happy to share it; even with all its faults.

In the future I will come back and resolve all the outstanding issues.

## 1.4. Disclaimers

### 1.4.1. The code was not written by an AI

**umar**

The compiler is only at ~12,000 lines of code and all the code was hand written by me. However, at the moment of writing this, it is undeniable that AI tools have changed the economic landscape of software engineering.

"Vibe Coding" a compiler is completely possible with the quality and feature set produced by an AI agent being better than I can produce in maybe an hour. This is true for any domain in software and, in general, any domain of work. The utility and speed of AI agents is undeniable. However, in doing so we will be no closer to understanding compiler construction and how runtime systems work.

I still believe there is something special that happens when deep human thought is applied to create solutions. In doing so we understand systems better and are more equipped to challenge conventional wisdoms, helping to produce innovative ideas.

### 1.4.2. The code explanations in this book was written mostly by an AI ( Claude-Code )

**umar**

Although the structure of the book was determined by me, the code explanations were created by asking Claude to explain my own source code. I initially attempted to write this book by hand, but given deadlines, this took a considerable amount of time.

I decided to use Claude Code to generate explanations of my own source code. This made it incredibly easy for me to paste in explanations that are formatted correctly and also generate visual representations of my code.

I must admit that once I did this, the book lost a personal flavour and style to it and has become more of a reference book. Although the explanations generated by Claude Code are impressive and make sense to me, they lack depth and I think if the book is used independently for study, it will not be a good resource. If this book does reach an audience, I would love to know what you think.

I encourage readers to watch the YouTube video that’s associated with this book: "Understanding Compilers and Runtimes - Type Checking, Generics, Virtual Threads, Garbage Collection"

### 1.4.3. How can the reader tell what sections were written by AI vs Umar Karimabadi (me)

**umar**

If text is contained in this grey boxed background with my name on the top left, it was written by me. Any text that does not have this background was written by Claude Code unless it is the source code of the compiler. You will find nearly the majority of the explanations, if not all, were written by an AI (Claude Code).

## 1.5. What this book does not cover

**umar**

Direct code generation is not supported in the noC compiler. noC compiles down to virtual stack based instruction set and the program is interpreted in a virtual machine. The more interesting subjects of register allocation and code scheduling will not be covered as these problems only arise if you support direct code generation.

In addition, compiler optimizations will not be covered in this book as I did not explore any material around this.

I initially wanted to support multiple instruction sets but I decided to abandon this pretty early on in the project beacuse of time constraints. I hope readers still find this publication useful.

# Chapter 2. High level overview of langauge features

## 2.1. Language Design

- Struct-based OOP (no classes or inheritance)
- Full Generic Support for struct definitions and functions
- Full functional programming support with closures and high level functions
- Integrated error handling as language primitives with `throws`, `try` and `defer`
- Virtual Threads as the basis for threading with non-blocking I/O

## 2.2. Data Types

- Primitives: `int`, `char`, `byte`, `bool`, `string`
- Arrays: Fixed-size (`T[size]`), multi-dimensional (`int[3][3][3]`), with `.len` property
- Structs: Custom data structures, support generics (`struct stack<T>`)
- Enums: `enum GC_LOG_LEVEL { VERBOSE, NONE }`
- `null`: For uninitialized object references
- Type inference for local variables

## 2.3. Control Flow

- `if`/`else` conditionals
- `for` loops (traditional and infinite with `break`)
- `match` statement for matching on enums
- `break` and `continue`

## 2.4. Functions

- Standard syntax: `function return_type name(params) { … }`
- First-class functions: Can be passed as parameters, stored in variables/arrays
- Function typedefs: `typedef function return_int :: () → int;`
- Nested functions and closures
- `throws`/`try`: `function name() throws { … }` and `try function_call()` for error handling

## 2.5. Memory Management

- `new` keyword for object/array allocation

- Automatic garbage collection

## 2.6. Threading & Async

- Virtual threads: `no function() { … }` launches async tasks
- non blocking awaitable futures with `await(future)`
- Reentrant `mutex` for synchronization (`lock()`, `try_lock()`, `unlock()`)

## 2.7. Non Blocking I/O

- Sockets: `socket`, `server_socket` for non blocking network I/O
- Files: `file_handle` with read/write operations
- Output: `printf()` for formatted printing

## 2.8. noC Simple Code Examples

> You can find bigger examples of `noC` code in the `noCdir`

**User Defined Structs**

```
struct err {
    string err_msg;
}
```

**Generic Structs**

```
struct stack<T> {
    T[] elements;
    int top_index;
}

struct array_list<T> {
    T[] arr;
    int num_elements;
}
```

**Generic Functions**

```
function init_stack(stack<T> stack) {
    stack.elements = new T[10];
    stack.top_index = 0;
}
```

```
function T pop(stack<T> stack) {
    if (is_empty(stack)) {
        return null;
    }
    stack.top_index--;
    return stack.elements[stack.top_index + 1];
}
```

### Lambda Functions

```
x = new int[5];

output = map(x, function(int x) int {
    return x * 2;
});
```

### Local Functions

```
function local_function_demo() {


    x = 1;


    function inner_function() {
        printf("The value of x is %d\n", x);
    }


    inner_function(); // prints the value of x is 1


}
```

### Closure Objects

```
function counter_object create_counter_closure() {
    counter = 1;  // Captured variable

    function increment_counter() {
        counter++;  // Closes over 'counter'
    }

    function decrement_counter() {
        counter--; // Closes over 'counter'
    }

    function print_x() {
        printf("The value of x is %d\n", counter);  //  Closes over 'counter'
    }
    counter_obj = new counter_object();
    counter_obj.increment = increment_counter;
    counter_obj.decrement = decrement_counter;
    counter_obj.print_x = print_x;
    return counter_obj;
}

closure = create_counter_closure();

increment_fn = clsoure.increment_counter;
increment_fn(); // incremnts the counter in the closure
```

### Exception Handling with Try Statements

```
// Function that can throw, must be tried or handled
function T array_list_at(array_list<T> list, int index) throws {
    if (index < 0 || index >= list.arr.len) {
        throw error("array_list attempting to access unsafe index");
    }
    return list.arr[index];
}

function test(array_list<T> list) throws {
    // The error is not handled but try'd
    value = try array_list_at(list, 0);
}

list = new array_list<int>();

err = test(list);  // error is explicitly handled
if (err != null) {
    print_err(err);
    return;
}
```

### Defer statements

```
function doWork(mutex mutex, BoxedInt counter) {
    lock(mutex);
    defer unlock(mutex);  // Will run when function exits

    for (i=0; i<10000; i++) {
        add(counter, 1);
    }
}  // unlock(mutex) executes here automatically
```

### No Routines (Coroutines) and Await

```
// Spawn a no routine (returns a future)
// The future has a result that is the return type of the funciton
result_future = no function() int {
    printf("Running on thread %d\n", get_vthreadid());
    sleep(2);
    return 1;
}

// Await the result
num = try await(result_future);
printf("The number is %d\n", num);
```

# Chapter 3. The phases in noC Compilation

**umar**

The main.cpp file describes in code clearly the compilation phases:

- Read files
- Tokenize and Parse Files
- Type Check Files
- Create Sizing information for objects
- Generate IR code
- Boot VM and Run IR code

The rest of this book will follow the compilation phases in order

*main.cpp*

```
        // Read file
        vector<string>* noc_files = NULL;
        noc_files = read_noc_files_from_current_dir();

        // Tokenize and Parse
        auto parse_unites = new vector<file_parse_unit*>();
        bool is_parsing_errors = false;
        for (auto nocfilename : *noc_files) {
            auto f_open_result = open_file(nocfilename);
            TokenStream* tokenStream = tokenize(nocfilename, f_open_result.buffer,
f_open_result.len);
            tokenStream->print_token_stream();
            file_parse_unit* file_unit = parseProgram(tokenStream);
            parse_unites->push_back(file_unit);

        }

        // Type check
        SymbolTable* symbolTable = createSymbolTable();
        queue<file_parse_unit*>* depedancy_order = create_depedancy_order(
parse_unites);
        while (!depedancy_order->empty()) {
            auto unit = depedancy_order->front();
            depedancy_order->pop();
            typeCheck(symbolTable, unit->program);
        }

        // Resolve closures and creating sizing information
        resolve_closures(symbolTable);
        configure_layout(symbolTable, RuntimeBackend::NOC_VIRTUAL);
```

```
    // Gen IR code
    auto entryPoint = gen(symbolTable);
    if (entryPoint->mainIp == -1) {
        throw runtime_error("main entry point cannot be found");
    }

    // Boot Virutal Machine Runtime and Run Code
    boot(entryPoint, symbolTable);
    return 0;
```

# Part 1 : Parsing and Type Checking

# Chapter 4. Tokenisation and Parsing

## 4.1. Tokenisation

**umar**

In classical compiler literature, there is significant content on parsing, grammar construction, and Tokenization/Lexing. With NFAs, DFAs, context-sensitive grammars, inductive derivation, and a whole host of mathematical formalizations. In college/university, a compiler course would invest a significant amount of time in creating a course that targets this specific literature only, for one reason: because it can be mathematically formalized, it can be examinable in a test.

However, we can simplify this entirely and ignore NFAs, DFAs, and all the mathematical jargon that is associated with it in favour of a handwritten Tokenizer and Parser. The latter is significantly simpler and requires no effort to understand the mathematical formalizations and the gymnastics associated with that.

The first phase of compilation is Tokenization. This is turning the actual programming text into Tokens that we can feed into the parsing stage. A Token effectively resembles a single word. Tokenization makes Parsing simpler as the Parsing now just focuses on a stream of Tokens instead of a stream of characters. The Tokenizer is responsible for creating chunks of text called Tokens.

## Output: The Token Stream

The tokenizer produces a `TokenStream` object containing a list/vector of `Tokens`. Each token carries:

- Its Token Type (e.g., `IDENTIFER`, `NUMBER_LITERAL`, `OPEN_CURLY_BRACKET`)
- The actual text from the source)
- Source location information (line and column numbers)

This token stream becomes the input to the next compilation phase: parsing.

```
enum TokenType {
    INT,
    BYTE,
    UINT,
    BOOL,
    DOUBLE,
    FLOAT,
    STRING_LITERAL_TOKEN,
    CHAR_LITERAL_TOKEN,
    CHAR,
    BREAK,
    CONTINUE,
    STRUCT,
    TYPE,
    NEW,
    ...
}

struct Token {
    TokenType tokenType;
    bool isTokenTypePrimitve;
    string literal;
    size_t colum_number;
    size_t line_number;
};
```

**Example**

The Tokenisation of the following program will produce the following TokenStream where each word in the progam is associated with a Token

```
function main() throws {
    x = 1;
}
```

```
TokenStream :: [FUNCTION,IDENTIFER, OPEN_ROUND_BRACKET, CLOSE_ROUND_BRACKET, THROWS,
OPEN_CURLY_BRACKET, IDENTIFER, EQUAL, NUMBER_LITERAL, SEMI_COLON, CLOSE_CURLY_BRACKET]
```

```
FUNCTION --------------------> function
IDENTIFIER ------------------> main
OPEN_ROUND_BRACKET --> (
CLOSE_ROUND_BRACKET > )
THROWS --------------------------> throws
OPEN_CURLY_BRACKET --> {
IDENTIFIER ------------------> x
EQUAL ----------------------------> =
NUMBER_LITERAL ----------> 1
SEMI_COLON ------------------> ;
CLOSE_CURLY_BRACKET > }
```

umar

Compiler construction naturally has very well-defined barriers that allow you to solve smaller problems, helping to build the final solution.

Clumping pieces of text together to produce Tokens makes the Parsing stage simpler as the parser no longer looks at individual characters but a list of Tokens.

Throughout this book we will see how stages of compilation occur with each layer complimenting the layer that comes after it to create code that we can execute and the interface that supports the runtime.

## Token Categories

The tokenizer recognizes several categories of tokens:

### Keywords and Primitive Types

Reserved words like `function`, `if`, `else`, `for`, `return`, `struct`, and primitive type names (`int`, `char`, `bool`, `double`, `float`, `byte`, `void`) are recognized through a keyword lookup table:

```
static const struct KeywordMapping {
    char* literal;
    size_t length;
    TokenType type;
    bool isPrimitive;
} keywords[] = {
    {NEW_LITERAL, LEN(NEW_LITERAL) - 1, NEW, false},
    {INT_LITERAL, LEN(INT_LITERAL) - 1, INT, true},
    // ... additional keywords
};
```

When the tokenizer encounters a sequence of letters, it first checks this table. If a match is found, the appropriate keyword token is emitted. Otherwise, the sequence is treated as an identifier.

### Single-Character Operators and Delimiters

Characters like `+`, `-`, `*`, `/`, `{`, `}`, `(`, `)`, `;`, and `,` are mapped using an ASCII lookup table for O(1) access:

```
static OperatorMapping* operatorAsciiMap[126];
```

Each entry maps a character’s ASCII value to its corresponding token type. This approach avoids linear searches through operator lists.

## Multi-Character Operators

Some operators consist of two characters: `==`, `!=`, `<=`, `>=`, `&&`, `||`, `++`, `--`, `->`, and `::`. The tokenizer handles these through lookahead:

1. Read the first character
2. Peek at the next character
3. Check if the two-character combination matches a known operator
4. If yes, consume both characters and emit the compound token
5. If no, emit only the single-character token

## Numeric Literals

Numbers are recognized by their leading digit. The tokenizer distinguishes between:

- **Integer literals** — Sequences of digits (e.g., `42`, `1000`)
- **Real literals** — Digits containing a decimal point followed by more digits (e.g., `3.14`)

The decimal point handling is noteworthy: when a dot is encountered after digits, the tokenizer peeks ahead. If another digit follows, it's a floating-point number. Otherwise, the dot is a separate token (for member access).

## String and Character Literals

String literals are delimited by double quotes (`"`). The tokenizer:

1. Records the opening position (for error reporting)
2. Consumes characters until the closing quote
3. Extracts the content between the quotes
4. Emits a `STRING_LITERAL_TOKEN`

Character literals use single quotes (`'`) and must contain exactly one character.

## Identifiers

Any sequence of letters (and underscores) that doesn't match a keyword becomes an identifier token. Identifiers typically represent variable names, function names, or type names.

# The Main Tokenization Loop

The core algorithm follows this pattern:

```
while (state != Done) {
    switch (state) {
    case ReadNormal:
        while (canRead(charStream)) {
            char character = peakChar(charStream);

            if (isCharSingleQuote(character)) {
                // Handle character literal
            }
            else if (isCharSpeechMark(character)) {
                // Handle string literal
            }
            else if (isCharALetter(character)) {
                // Handle keyword or identifier
            }
            else if (isCharacterOperator(character)) {
                // Handle operator (single or multi-character)
            }
            else if (isCharANumber(character)) {
                // Handle numeric literal
            }
            else if (isNotTokenizable(character)) {
                // Skip whitespace
            }
            else {
                // Emit error for unrecognized character
            }
        }
        break;

    case Comment:
        // Consume characters until newline
        break;
    }
}
```

# 4.2. Parsing

**umar**

`Tokenisation` and `Parsing` are trying to rebuild the program from a text representation found in our source code files into a programmable representation that we can manipulate. The most intuitive representation is the AST (Abstract Syntax Tree).

Parsing is the stage of taking the token stream and rebuilding an explicit tree representation of our source code (the AST). Each construct in our programming language is represented by a node in the tree.

We have a node type for each construct in our programming language.

The parser consumes the token stream and builds an Abstract Syntax Tree (AST). It uses recursive descent parsing, where each grammar rule corresponds to a parsing function.

## 4.2.1. The Token Stream Interface

The parser interacts with tokens through peek/read operations:

```
Token* TokenStream::peakToken() {
    if (!this->canRead()) {
        return &endOfStream;  // Sentinel for end-of-input
    }
    return this->tokenStream->at(this->position);
}

Token* TokenStream::readToken() {
    Token* token = tokenStream->at(this->position);
    this->position++;
    return token;
}
```

## 4.2.2. Top-Level Parsing Loop

The main parser function processes top-level declarations:

```
file_parse_unit* parseProgram(TokenStream* tokenStream) {
    AstBody* body = createAstBody();
    file_parse_unit* unit = create_file_parse_unit(tokenStream->file_name);
    unit->program = body;

    while (tokenStream->canRead()) {
        Token* token = tokenStream->peakToken();

        switch (token->tokenType) {
```

```
        case IMPORT:
            // Parse import statement
            parseImport(tokenStream, unit->imports);
            break;

        case FUNCTION:
            // Parse function definition
            AstFunctionDefinition* functionDef = parseFunctionDecleration(
tokenStream);
            addStatementToBody(body, functionDef);
            break;

        case EXTERNAL:
            // Parse external function declaration
            AstExternalFunction* externalfunction = parseExternalFunction(
tokenStream);
            addStatementToBody(body, externalfunction);
            break;

        case STRUCT:
            // Parse struct definition
            AstStructNode* strct = parseStructDecleration(tokenStream);
            addStatementToBody(body, strct);
            break;

        case ENUM:
            // Parse enum definition
            AstEnum* enumNode = parseEnumDecleration(tokenStream);
            addStatementToBody(body, enumNode);
            break;

        case TYPEDEF:
            // Parse type definition
            AstFunctionTypeDef* typeDef = parseFunctionTypeDef(tokenStream);
            addStatementToBody(body, typeDef);
            break;
        }
    }

    return unit;
}
```

*Parsing Flow Visualization*

```
Token Stream: [FUNCTION, IDENTIFER("add"), OPEN_ROUND_BRACKET, ...]
                  |
                  v
         +-----------------+
         | parseProgram()  |
         +-----------------+
                  |
                  | sees FUNCTION token
                  v
    +---------------------------+
    | parseFunctionDecleration()|
    +---------------------------+
          |         |        |
          v         v        v
    parseName  parseParams  parseBody
          |         |        |
          v         v        v
    "add"      [(x:int)]   [statements]
```

## 4.3. Visualizing the AST

### 4.3.1. A Simpler Example: Void Function

Consider this simple function:

```
function greet() {
    printf("Hello");
}
```

The AST looks like:

```
AstFunctionDefinition
 ├──── functionName: "greet"
 ├──── parameters: []
 ├──── returnIsVoid: true
 ├──── throwsError: false
 └──── body: AstBody
     └──── statements:
         └──── AstFunctionCall
             ├──── name: "printf"
             └──── arguments:
                 └──── AstLiteralValue
                     ├──── valueType: StringLiteral
                     └──── literalValue: "Hello"
```

A more complex example with arithmetic:

```
function int add(int a, int b) {
    return a + b;
}
```

```
AstFunctionDefinition
 ├──── functionName: "add"
 ├──── parameters:
 │    ├──── AstDecleration { name: "a", type: "int" }
 │    └──── AstDecleration { name: "b", type: "int" }
 ├──── returnTypeDecl: AstTypeDecleration { typeStr: "int" }
 ├──── returnIsVoid: false
 └──── body: AstBody
     └──── statements:
         └──── AstReturnStatement
             └──── expression: AstArithMeticExpression
                 ├──── op: "+"
                 ├──── left: AstIdentifier { name: "a" }
                 └──── right: AstIdentifier { name: "b" }
```

### 4.3.2. The AST Node Structure

All AST nodes inherit from a base structure:

```
struct AstNode {
    NodeType type;        // Discriminator for node kind
    AstNode* sub;         // Substitution field for transformations
    int line_number;      // Source location for errors
    int column_number;
    string file_name;
};
```

The NodeType enum covers all possible node kinds:

```
enum NodeType {
    // Definitions
    StructDef, FunctionDefinition, ExternalFunctionDefinition,
    Interface, Enum, TypeDecleration,

    IfStatement, ForLoop, Assignment, Return, Defer, Try, Throw,
    BreakStatement, ContinueStatement, Match, Case,

    FunctionCall, Literal, ArithmeticExpression, BooleanExpression,
    Identifier, ArrayAccess, DotAccess, ObjectCreation, Null,

    // Closures and concurrency
    ClosureCreation, LambdaCall, AnonymousFunction, NoRoutine,

    // And more...
};
```

Key AST node types:

```
// Function definition
struct AstFunctionDefinition : public AstNode {
    string functionName;
    vector<AstDecleration*>* paramaters;
    AstTypeDecleration* returnTypeDecl;
    bool returnIsVoid;
    bool throwsError;
    bool hasPolyArgs;     // Has polymorphic type parameters
    bool isExternal;
    AstBody* body;
};

// Struct definition
struct AstStructNode : public AstNode {
    bool isDefinedAsPoly;
    vector<string>* polyHeader;  // Generic type parameters
    string name;
    vector<AstDecleration*>* members;
};

// If statement with optional else-if chains
struct AstIfStatement : public AstNode {
    AstNode* booleanExpression;
    AstBody* ifbody;
    vector<AstIfElse*>* ifElses;  // else-if chains
    AstBody* elseBody;            // Final else (optional)
};

// Binary expressions
struct AstArithMeticExpression : public AstNode {
    string op;        // "+", "-", "*", "/", "%"
    AstNode* left;
    AstNode* right;
};

struct AstBooleanExpression : public AstNode {
    string op;        // "&&", "||", "==", "!=", "<", ">", etc.
    AstNode* left;
    AstNode* right;
};
```

**umar**

I hope this demonstrates that we are effectively transforming the source code text into explicit AST nodes, with each node explicitly capturing a construct (for, if, struct, etc.) that is supported by the programming language.

With the Tree structure explicitly represented, we can traverse through the nodes. This is often called "Tree Walking".

I would encourage the user to read the `parser.cpp` to see how the AST Nodes are created for more complicated constructs.

*All the Ast Nodes that are supported in the langauge, each NodeType referes to some programming construct supported in the langauge.*

```
enum NodeType {
    StructDef,
    FunctionCall,
    ClosureCreation,
    FunctionTypeDef,
    LambdaCall,
    FunctionParam,
    AnonymousFunction,
    FunctionDefinition,
    PolyClonedFunctionDefinition,
    ExternalFunctionDefinition,
    Interface,
    IfStatement,
    Literal,
    ArithmeticExpression,
    BooleanExpression,
    Identifier,
    TypedIdentifier,
    Body,
    Assignment,
    BreakStatement,
    ContinueStatement,
    ForLoop,
    ArrayAccess,
    ArraySubscriptAssignment,
    Return,
    ObjectCreation,
    DotAccess,
    DotAssignment,
    Null,
    Defer,
    Try,
    TryLabel,
    Throw,
    Tuple,
    Match,
    Case,
    Enum,
    TypeDecleration,
    ScopedIdentifier,
    Yield,
    NoRoutine,
    CoRoutineLifted,
    TryCatchAssignment,
    NegativeExpression
};
```

# Chapter 5. Type checking

**umar**

Type checking is a recursive "Tree walk" across the Abstract Syntax Tree. The type for a node is known immediately if it is a literal (string, bool, etc.) or has to be inferred by recursively walking non-literal nodes (FunctionCalls, Arithmetic expressions, BooleanExpression, etc.).

As the Type Checker resolves symbols (identifier, function definition, struct), it creates an association with a Type.

Before we dive into examples about how the Type Checking works, we have to first understand how Types are represented in the compiler.

## 5.1. Types

The type system uses an inheritance hierarchy the represents types as a tree like structure

## Type Hierarchy

*Type Heirachy*

```
enum struct TypeOfType {
    BaseType,
    ArrayType,
    PolyType,
    EnumType,
    StructPolyType,
    LambdaType
};

struct Type {
    TypeOfType type_of_type;
    virtual string toString() = 0;
};

struct EnumType : public Type {
    string enumName;
};

struct BaseType : public Type {
    bool isPrimitive;
    string typeStr;
};

struct PolyType : public Type {
    string polyStr;
};
```

```
struct ArrayType : public Type {
    ArraySymbol* arrSymbol;
    Type* baseType;
};

struct StructPolyMorphicType : public Type {
    Type* baseType;
    vector<Type*>* polyTypes;
};

struct LambdaType : public Type {
    Type* returnType;
    vector<Type*>* arguments;
    string typeDefName;
};

struct TupleType : public Type {
    Type* type_1;
    Type* type_2;
};
```

```
Type (abstract base)
 ├──── BaseType              (primitives + struct instances)
 ├──── EnumType              (enum types)
 ├──── PolyType              (type variables like T)
 ├──── ArrayType             (multi-dimensional arrays)
 ├──── StructPolyMorphicType (generic instantiations like list<int>)
 ├──── LambdaType            (function types)
 └──── TupleType             (pairs of types)
```

# Each Type Explained

`BaseType` Represents primitive types (`int`, `double`, `bool`, `char`, `byte`, `void`) and non-generic struct instances. The `isPrimitive` flag distinguishes primitives from struct types. `typeStr` holds the type name (e.g., `"int"`, `"Person"`).

`EnumType` Represents user-defined enum types. Just holds the `enumName` since enums are simple named types.

`PolyType` Represents unbound type variables like `[T]` in generic definitions. `polyStr` holds the variable name (e.g., `"T"`). Used during type checking before concrete types are substituted.

`ArrayType` Represents multi-dimensional arrays. `num_dimensions` tracks dimensionality (1D, 2D, etc.). `baseType` points to element type (can be any `Type*`, enabling nested arrays).

`StructPolyMorphicType` Represents instantiated generic structs like `list<int>` or `map<string, int>`. `baseType` is the generic struct's base type. `polyTypes` is the list of concrete type arguments.

`LambdaType` Represents function/lambda types. `returnType` and `arguments` define the function signature. `typeDefName` allows named function type aliases.

`TupleType` Represents a pair of types. Simple two-element tuple with `type_1` and `type_2`.

This design allows us to create nested types like `list<map<string, int[]>>`

The type `list<map<string, int[]>>` is represented as a nested tree of `Type` structures:

```
StructPolyMorphicType (type_of_type = StructPolyType)
 ├──── baseType: BaseType { typeStr = "list", isPrimitive = false }
 └──── polyTypes: vector<Type*> [1 element]
     └──── [0]: StructPolyMorphicType (type_of_type = StructPolyType)
             ├──── baseType: BaseType { typeStr = "map", isPrimitive = false }
             └──── polyTypes: vector<Type*> [2 elements]
                 ├──── [0]: BaseType { typeStr = "string", isPrimitive = false }
                 └──── [1]: ArrayType (type_of_type = ArrayType)
                         ├──── num_dimensions: 1
                         └──── baseType: BaseType { typeStr = "int", isPrimitive =
true }
```

When Comparing Types we just compare the structure of the Type Tree rescursively. This comparision is the basis for type checking

## Checking Type Equality

Type equality is done by recursively traversing a Type tree structure againts another Type

```
bool areReturnTypesEqual(Type * returnType_1, Type * returnType_2) {

    if (returnType_1->type_of_type != returnType_2->type_of_type) {
        return false;
    }

    if (returnType_1->type_of_type == TypeOfType::BaseType) {

        BaseType* returnType1_as_basic = (BaseType*)returnType_1;
        BaseType* returnType2_as_basic = (BaseType*)returnType_2;

        return returnType1_as_basic->typeStr == returnType2_as_basic->typeStr;
    }

    else if (returnType_1->type_of_type == TypeOfType::ArrayType) {
        // Check base dimensions
        ArrayType* returnType1_as_arr = (ArrayType*)returnType_1;
        ArrayType* returnType2_as_arr = (ArrayType*)returnType_2;

        if (returnType1_as_arr->num_dimensions != returnType2_as_arr->num_dimensions)
{
            return false;
        }
        return areReturnTypesEqual(returnType1_as_arr->baseType, returnType2_as_arr-
>baseType);
    }
    else if (returnType_1->type_of_type == TypeOfType::EnumType) {

        EnumType* returnType1_as_enum = (EnumType*)returnType_1;
        EnumType* returnType2_as_enum = (EnumType*)returnType_2;
        return returnType1_as_enum->enumName == returnType2_as_enum->enumName;
    }
    ....
```

The function leverages the type tree in two key ways:

The `type_of_type` field in the base `Type` struct acts as a tag. The function first checks if both types have the same tag—if not, they cannot be equal.

```
if (returnType_1->type_of_type != returnType_2->type_of_type) {
    return false;
}
```

Once the tags match, the function casts the base `Type*` pointers to the appropriate derived type and compares their specific fields.

`BaseType`

Compares the `typeStr` field directly. Two base types are equal if they have the same name (e.g., both are `"int"` or both are `"Person"`).

```
return returnType1_as_basic->typeStr == returnType2_as_basic->typeStr;
```

`EnumType`

Compares the `enumName` field. Two enum types are equal if they share the same enum name.

```
return returnType1_as_enum->enumName == returnType2_as_enum->enumName;
```

`PolyType`

Compares the `polyStr` field. Two polymorphic type variables are equal if they have the same name (e.g., both are `T`).

```
return returnType1_as_poly->polyStr == returnType2_as_poly->polyStr;
```

`ArrayType`

First checks that dimensions match, then **recursively** compares the base types. This is where the tree structure becomes essential—an `int[][]` must match another `int[][]`, not an `int[]` or `double[][]`.

```
if (returnType1_as_arr->num_dimensions != returnType2_as_arr->num_dimensions) {
    return false;
}
return areReturnTypesEqual(returnType1_as_arr->baseType, returnType2_as_arr-
>baseType);
```

`LambdaType`

Recursively compares the return type, then checks argument count, then recursively compares each argument type. A function `(int, string) → bool` only equals another `(int, string) → bool`.

```
if (!areReturnTypesEqual(returnType1AsFunction->returnType, returnType2AsFunction-
>returnType)) {
    return false;
}
if (returnType1AsFunction->arguments->size() != returnType2AsFunction->arguments-
>size()) {
    return false;
}
for (size_t i = 0; i < returnType1AsFunction->arguments->size(); i++) {
    if (!areReturnTypesEqual(returnType1AsFunction->arguments->at(i),
                             returnType2AsFunction->arguments->at(i))) {
        return false;
    }
}
return true;
```

`StructPolyMorphicType`

Recursively compares the base struct type, then checks that the number of type arguments match, then recursively compares each type argument. A `list<int>` only equals another `list<int>`, not a `list<string>` or `map<int>`.

```
if (!areReturnTypesEqual(returnType1AsStructPoly->baseType,
                         returnType2AsStructPoly->baseType)) {
    return false;
}
if (returnType1AsStructPoly->polyTypes->size() != returnType2AsStructPoly-
>polyTypes->size()) {
    return false;
}
for (size_t i = 0; i < returnType1AsStructPoly->polyTypes->size(); i++) {
    if (!areReturnTypesEqual(returnType1AsStructPoly->polyTypes->at(i),
                             returnType2AsStructPoly->polyTypes->at(i))) {
        return false;
    }
}
return true;
```

This design allows arbitrarily nested types like `list<map<string, int[]>>` to be compared correctly—the function walks down both type trees in lockstep, verifying structural equality at every node.

*Example areReturnTypesEqual*

```
areReturnTypesEqual(list<int>, list<int>)
 │
 ├── Check: both are StructPolyType ✓
 ├── areReturnTypesEqual(list, list)     ← base types
 │   └── Check: both are BaseType, typeStr == "list" ✓
 └── areReturnTypesEqual(int, int)       ← poly type args
     └── Check: both are BaseType, typeStr == "int" ✓

Result: true
```

umar

**Is the Type Tree even needed?**

It had occurred to me as I was writing the book that the Type Tree is implicitly available within the AstNode. The Type Tree is just an explicit mapping/copy from the AstNode to the Type Tree structure effectively duplicating the AST.

This has led me to think whether the type tree is even needed as type equality could be just as easily determined by looking at the AstNode Structure.

However, I will pretend I never came across this realization as the change to the TypeChecker would be substantial; everything works with Type Objects. This could be an excellent exercise for an AI agent to see how well it performs with a more complex rewrite task like this. I have no doubt that Claude Code is up to the challenge!

## 5.2. The Symbol Table

The symbol table is the central data structure for type checking:

```
struct SymbolTable {
    FunctionSymbolTable* fSymbolTable;     // All functions
    StructSymbolTable* structSymbolTable;  // All struct layouts
    EnumSymbolTable* enumSymbolTable;      // All enums

    map<string, Symbol*>* globalSymbolTable;  // Global variables

    // Type information for each node (used by code gen)
    map<FunctionSymbol*, NodeTypeMap*>* nodeTypeMap;

    // Functions that escape their scope (become closures)
    vector<EscapingFunctionSymbol*>* functionDefThatEscape;
};
```

## 5.3. Symbol Table

A `Symbol` represents a named identifier in the program:

```
struct Symbol {
    int function_depth_level;   // Nesting level (0 = global)
    SymbolType symbolType;      // What kind of symbol
    string name;
    Type* returnType;           // The symbol's type
    Scoping_info scoping_info;  // For closure analysis
};

enum struct SymbolType {
    Lambda,              // A closure/lambda
    LambdaTypeDefinition,// typedef for function types
    GlobalType,          // Global variable
    Identifier,          // Local variable
    Struct,              // Struct type
    FunctionDefinition,  // Function
    Enum                 // Enum type
};
```

`FunctionSymbol` contains everything about a function:

```
struct FunctionSymbol {
    bool needsPolyBinding;        // Has unresolved generics
    bool isLocal;                 // Nested function
    bool needsLambdaLifting;      // Captures outer variables
    int functionDepthLevel;       // Nesting depth
    string name;
    size_t numberOfArgs;
    vector<Symbol*>* params;      // Parameter symbols
    Type* returnType;
    bool isExternal;              // Defined externally
    bool markedAsThrows;          // Can throw errors
    bool is_recursive;            // Self-recursive

    AstFunctionDefinition* functionDef;  // AST node
    AstBody* functionBody;
    PolyEnv* polyEnv;             // Generic bindings
    LambdaType* lambdaType;       // Function type
    FunctionSymbol* parent;       // Enclosing function

    // Non-local variables captured from outer scopes
    vector<NonLocalIdentifier*>* nonLocalIdentifiers;
};
```

### 5.3.1. Deriving Type from AstStructNode

Struct layouts are computed during type checking:

```
struct StructLayout {
    int identifier_index;           // Unique ID for runtime
    size_t total_size_of_struct;    // Memory size
    vector<string>* polyArgHeader;  // Generic parameters
    vector<Field*>* fields;         // Field information
};

struct Field {
    string name;
    Type* returnType;
    size_t mem_offset;   // Byte offset within struct
};
```

### 5.3.2. Example: Deriving Type from struct Person

```
struct Person {
    string name;
    int age;
}
```

This produces:

```
StructLayout {
    identifier_index: 1,
    total_size_of_struct: 16,   // 8 bytes each (pointers)
    polyArgHeader: [],
    fields: [
        Field { name: "name", type: "string", mem_offset: 0 },
        Field { name: "age",  type: "int",    mem_offset: 1 }
    ]
}
```

*Memory Layout*

```
Person object in memory:
+------------+------------+
| fields[0]  | fields[1]  |
| (name ptr) | (age ptr)  |
+------------+------------+
  offset 0     offset 1
```

## 5.4. Loading Function Symbols

Before type checking bodies, all function signatures are loaded:

```
void loadFunctionSymbols(SymbolTable* symTable, AstBody* program) {
    for (AstNode* node : *program->statements) {
        if (node->type == NodeType::FunctionDefinition) {
            AstFunctionDefinition* funcDef = (AstFunctionDefinition*)node;

            FunctionSymbol* fSym = new FunctionSymbol();
            fSym->name = funcDef->functionName;
            fSym->numberOfArgs = funcDef->paramaters->size();
            fSym->functionDepthLevel = 0;  // Top-level
            fSym->markedAsThrows = funcDef->throwsError;

            // Derive return type
            if (funcDef->returnIsVoid) {
                fSym->returnType = &voidReturn;
            } else {
                fSym->returnType = deriveType(funcDef->returnTypeDecl);
            }

            // Derive parameter types
            fSym->params = new vector<Symbol*>();
            for (AstDecleration* param : *funcDef->paramaters) {
                Symbol* pSym = new Symbol(param->decl_name,
                                          deriveType(param->typeDecl));
                fSym->params->push_back(pSym);
            }

            symTable->fSymbolTable->table->insert({fSym->name, fSym});
        }
    }
}
```

## 5.5. Type Checking Examples

## 5.6. IfStatement Type Checking

## 5.7. AST Structure

An if statement has this AST shape:

```
struct AstIfStatement : public AstNode {
    AstNode* booleanExpression;     // The condition
    AstBody* ifbody;                // Then branch
    vector<AstIfElse*>* ifElses;    // else-if chains
    AstBody* elseBody;              // Final else (optional)
};
```

## 5.8. Type Checking Process

## 5.9. Step-by-Step Breakdown

### 5.9.1. Step 1: Validate the if Condition

The condition expression must evaluate to a valid type:

```
Type* conditionType = typeCheckExpression(symTable, fSymbol,
                                           ifStmt->booleanExpression);
```

### 5.9.2. Step 2: Enforce Boolean Type

The condition must be boolean:

```
if (!isReturnTypeBool(conditionType)) {
    emit_error_info(ifStmt->booleanExpression,
                    "If condition must be boolean, got: " +
                    conditionType->toString());
}
```

### 5.9.3. Step 3: Recursively Type Check Bodies

Both branches are checked in the current scope:

```
typeCheckBody(symTable, fSymbol, ifStmt->ifbody);

if (ifStmt->elseBody != nullptr) {
    typeCheckBody(symTable, fSymbol, ifStmt->elseBody);
}
```

### 5.9.4. Step 4: Handle else if Chains

Each else-if has its own condition and body:

```
for (AstIfElse* elseIf : *ifStmt->ifElses) {
    Type* elseIfCondType = typeCheckExpression(symTable, fSymbol,
                                                elseIf->booleanExpression);
    if (!isReturnTypeBool(elseIfCondType)) {
        emit_error_info(elseIf->booleanExpression,
                        "Else-if condition must be boolean");
    }
    typeCheckBody(symTable, fSymbol, elseIf->ifbody);
}
```

## 5.10. Type Checking Function Call

```
Type* typeCheckFunctionCall(SymbolTable* symTable,
                            FunctionSymbol* fSymbol,
                            AstFunctionCall* call) {
```

### 5.10.1. Step 1: Retrieve Function Symbol

```
    FunctionSymbol* callee = getFunctionSymbol(symTable, call->name);
    if (callee == nullptr) {
        emit_error_info(call, "Unknown function: " + call->name);
        return nullptr;
    }
```

### 5.10.2. Step 2: Validate Argument Count

```
    if (call->arguments->size() != callee->numberOfArgs) {
        emit_error_info(call,
            "Expected " + to_string(callee->numberOfArgs) +
            " arguments, got " + to_string(call->arguments->size()));
    }
```

### 5.10.3. Step 3: Type Check Each Argument

```
    for (size_t i = 0; i < call->arguments->size(); i++) {
        AstNode* arg = call->arguments->at(i);
        Type* argType = typeCheckExpression(symTable, fSymbol, arg);
        Type* paramType = callee->params->at(i)->returnType;

        if (!typesCompatible(argType, paramType)) {
            emit_error_info(arg,
                "Argument " + to_string(i) + " type mismatch: " +
                "expected " + paramType->toString() +
                ", got " + argType->toString());
        }
    }

    return callee->returnType;
}
```

## 5.11. Type Flow by Node Category

*Table 1. Type Inference by Node Type*

| Node Type | Input Types | Result Type |
|---|---|---|
| ArithmeticExpression | left: numeric, right: numeric | Wider of (left, right) |
| BooleanExpression | left: any, right: any (comparable) | bool |
| FunctionCall | Arguments match parameters | Function return type |
| ArrayAccess | array[int] | Array element type |
| DotAccess | struct.field | Field type |
| ObjectCreation | new TypeName | TypeName |
| Literal | (none) | Literal type (int/bool/string/etc) |

## Visual Flow

*Example for Type Checking Assignment*

```
                    Assignment
                    x::int = add(1, 2)
                        |
                        v
        +------------------------------+
        | Infer type of right-hand side|
        +------------------------------+
                        |
                        v
                    FunctionCall
                    add(1, 2)
                        |
        +--------------+--------------+
        v                             v
    Argument 0                    Argument 1
    Literal: 1                    Literal: 2
        |                             |
        v                             v
    infer -> int                  infer -> int
        |                             |
        +--------------+--------------+
                        v
        +------------------------------+
        | Check: int == expected param |
        | Check: int == expected param |
        | Return: add's return type    |
        +------------------------------+
                        |
                        v
                    returns int
                        |
                        v
        +------------------------------+
        | Check: int == declared x::int|
        | [OK] Assignment valid        |
        +------------------------------+
```

# 5.12. Generics and Polymorphism

> **umar**
>
> Generics and Polymorphism are resolved by creating a complete clone of the Generic struct or the Polymorphic function with types substituted. It is a literal clone of the AST creating a new struct definition or a function definition.

### What is PolyEnv?

PolyEnv (Polymorphic Environment) is a data structure that maps polymorphic type variables to their concrete types during type checking. It serves as the binding context when instantiating generic functions or structs with specific types.

```
struct PolyEnv {
    int num_unresolved_polyTypes;    // Tracks how many poly types remain unbound
    map<string, Type*>* env;         // The mapping: e.g., {"T" -> int, "B" -> string}
};
```

The num_unresolved_polyTypes counter is crucial: it tracks whether a full monomorphization can occur or if the expansion must remain partially polymorphic.

# 5.13. The Two-Phase Algorithm: Merge then Unify

The polymorphic expansion process follows a two-phase algorithm:

## 5.13.1. Phase 1: Merging (Building the PolyEnv)

The mergeTypes() function walks two type graphs in parallel—one generic, one concrete—and populates the PolyEnv with bindings as it discovers correspondences.

```
merge(list[T], list[int]) → builds PolyEnv: {T → int}
```

How merging works:

1. Poly types (T, U, etc.): This is where bindings are created:
   - If the poly variable isn't in the env yet, bind it to the concrete type
   - If it's already bound, verify the new concrete type matches the existing binding (consistency check)
   - If the concrete type is also polymorphic, increment num_unresolved_polyTypes
2. Compound types (arrays, structs, lambdas): Recursively merge their components:
   - Arrays: merge base types, verify dimensions match
   - Polymorphic structs: merge base type and each poly type parameter

    - Lambda types: merge return type and each argument type

### 5.13.2. Phase 2: Unification (Applying the PolyEnv)

The unifyTypes() function takes a polymorphic type and the populated PolyEnv, then substitutes all poly variables with their concrete bindings.

```
Given PolyEnv: {T → int}
unify(list[T]) → list[int]
```

How unification works:

1. Base types: Return unchanged (already concrete)
2. Poly types: Look up in env and return the bound concrete type
3. Arrays: Unify the base type, create new array type with concrete base
4. Structs: Unify each poly type parameter
5. Lambdas: Unify return type and all argument types

## 5.14. Expanding a Polymorphic Function

The expandPoly() function for provides the full expansion process:

```
expandPoly(symbolTable, polyFunctionSymbol, functionCallArgTypes, polyEnv,
frame_scope)
```

```
 Clone the function body: The cloneToNonPoly() function creates a shallow clone of the
AST where only nodes requiring type substitution are deep-cloned:
- Object creations (need concrete struct types) and any poly morphic object creationg
are replaced with the unified type in the PolyEnv
```

## 5.15. Example Flow

Given:

```
function T identity(T x) {
    return x
}

identity(42)  // call site
```

1. At call site, expandPoly is invoked with functionCallArgTypes = [int]
2. Merge: The PolyEnv is built up. T □ int → PolyEnv: {T → int}

3. Unify return type: T → int
4. Clone the entire function usign the PolyEnv to replace generic definitons
5. Result: cloned function

```
function int identity(x: int) {
    return x
}
```

## 5.16. Visualization: Polymorphic Function Resolution

The following ASCII diagram shows the complete process of resolving `identity(42)`:

```
                         DEFINITION TIME
                         ==============
    +-----------------------------------------------------------+
    |  function T identity(T x) { return x; }                   |
    +-----------------------------------------------------------+
                              |
                              v
    +-----------------------------------------------------------+
    |  FunctionSymbol: "identity"                               |
    |  +-- params: [Symbol { name: "x", type: PolyType("T") }]  |
    |  +-- returnType: PolyType("T")                            |
    |  +-- needsPolyBinding: true    <-- marked as generic      |
    +-----------------------------------------------------------+


                           CALL TIME
                           =========
    +-----------------------------------------------------------+
    |  identity(42)                                             |
    +-----------------------------------------------------------+
                              |
                              v
    +-----------------------------------------------------------+
    |  Type check argument: 42 -> int                           |
    |  functionCallArgTypes = [int]                             |
    |  Detect: needsPolyBinding = true                          |
    |  -> Call expandPoly(identity, [int], polyEnv)             |
    +-----------------------------------------------------------+


                        PHASE 1: MERGE
                        ==============
         Build PolyEnv by matching generic <-> concrete

    Generic Param          Concrete Arg           PolyEnv
```

```
+-------------+       +-------------+       +---------------+
| T           |  <->   | int         |  --->  | "T" -> int    |
+-------------+       +-------------+       +---------------+

mergeTypes(PolyType("T"), BaseType("int"), polyEnv)
  +--> polyEnv.env["T"] = int
  +--> return int


                   PHASE 2: UNIFY
                   ==============
    Apply PolyEnv to resolve return type

Generic Return         PolyEnv                Concrete Return
+-------------+        +---------------+      +---------------+
| T           |   +    | "T" -> int    | ---> | int           |
+-------------+        +---------------+      +---------------+

unifyTypes(PolyType("T"), polyEnv)
  +--> lookup polyEnv["T"] = int
  +--> return int


               PHASE 3: SPECIALIZE
               ====================
       Create new mangled function

+-----------------------------------------------------------+
|  New FunctionSymbol: "identity_int_int"                   |
|  +-- params: [Symbol { name: "x", type: int }]            |
|  +-- returnType: int                                      |
|  +-- needsPolyBinding: false   <-- fully concrete         |
|  +-- functionBody: cloned AST with types substituted      |
+-----------------------------------------------------------+

Name mangling: "identity" + "_int" (param) + "_int" (return)
               +------------------------------------------+
                          "identity_int_int"


               PHASE 4: CLONE AST
               ==================
     Clone function body with type substitutions

Original AST                      Cloned AST
+-----------------------+         +-----------------------+
| AstReturn             |         | AstReturn             |
| +-- AstIdentifier     |  --->   | +-- AstIdentifier     |
|     name: "x"         |         |     name: "x"         |
|     type: T           |         |     type: int         |
+-----------------------+         +-----------------------+
                    FINAL RESULT
                    ============

Before expansion:                 After expansion:
+-----------------------+         +---------------------------+
| function T identity   |         | function int              |
|   (T x) {             |  --->   |   identity_int_int        |
|   return x;           |         |   (int x) {               |
| }                     |         |   return x;               |
|                       |         | }                         |
| identity(42)          |         | identity_int_int(42)      |
+-----------------------+         +---------------------------+

Symbol Table now contains:
+-----------------------------------------------------------+
| "identity"         -> FunctionSymbol (generic, template)  |
| "identity_int_int" -> FunctionSymbol (specialized)        |
+-----------------------------------------------------------+
```

The key insight is that polymorphic functions are **templates** that get specialized at each unique call site. The specialized version is cached in the symbol table, so calling `identity(42)` again will reuse `identity_int_int` instead of creating another copy.

# Part 2 : Code Generation

# Chapter 6. Virtual stack based instruction set

**umar**

The virtual machine instruction set is very similar to what the JVM instruction set provides and was used as reference when creating different Opcodes.

Code generation produces an instruction set that is held only in memory. Unlike Java or other systems, the generated code is not persisted to disk so cannot be loaded at runtime. The persistence is simple enough to do but just decided not to add this.

Stack-based virtual machines are incredibly trivial to generate code for. Operands/arguments are pushed onto the thread's stack and depending on the opcode, an implied number of operands are popped from the stack to execute the Opcode. A simple preorder of the AST where you visit the children first and then generate code for the parent node.

## 6.1. Instruction Format

Each instruction has an opcode and optional operands:

```
struct Instruction {
    Opcode op_code;
    string symbol;   // Optional symbolic reference
    int arg1;
    int arg2;
};
```

## 6.2. Instruction Categories

### 6.2.1. Stack Operations

```
LOAD_I symbol      - Push local variable onto stack
STORE symbol       - Pop stack and store in local variable
LOAD_CONST idx     - Push constant from pool
LOAD_IMMEDIATE val - Push immediate integer value
LOAD_NULL          - Push null object
```

### 6.2.2. Arithmetic Operations

```
ADD         - Pop two, push sum
SUBTRACT    - Pop two, push difference
MULTIPLY    - Pop two, push product
```

```
DIVIDE     - Pop two, push quotient
MOD        - Pop two, push remainder
```

*Stack Effect of ADD*

```
Before:      After:
+-----+      +-----+
|  5  |      |  8  |
+-----+      +-----+
|  3  |      | ... |
+-----+
```

### 6.2.3. Type Conversions

```
D2I    - Double to Int
I2D    - Int to Double
C2I    - Char to Int
I2C    - Int to Char
```

### 6.2.4. Comparison Operations

```
LESS_THAN         - Pop two, push (a < b)
LESS_THAN_EQUAL   - Pop two, push (a <= b)
MORE_THAN         - Pop two, push (a > b)
MORE_THAN_EQUAL   - Pop two, push (a >= b)
EQUAL             - Pop two, push (a == b)
NOT_EQUAL         - Pop two, push (a != b)
AND               - Pop two, push (a && b)
OR                - Pop two, push (a || b)
```

### 6.2.5. Control Flow

```
JUMP offset        - Unconditional jump
JUMP_IF_FALSE off  - Pop boolean, jump if false
LABEL              - No-op marker for jump targets
```

### 6.2.6. Function Calls

```
PUSH_ARG           - Push value for function argument
POP_ARG symbol     - Pop argument into parameter slot
CALL name, nargs   - Call function, create frame
CALL_LAMBDA        - Call closure/lambda
CALL_WITH_TRY_CATCH- Call that may throw
RETURN             - Return with value
```

```
RETURN_FROM_VOID   - Return without value
```

### 6.2.7. Object Operations

```
NEW type_idx       - Allocate new struct instance
NEW_ARR            - Allocate new array
LOAD_FIELD offset  - Pop object, push field value
STORE_FIELD offset - Pop value and object, store field
LOAD_ARR_I         - Array index load
STORE_ARR_I        - Array index store
LOAD_ARR_LEN       - Get array length
```

### 6.2.8. Error Handling

```
THROW              - Throw exception
DEFER              - Register deferred call
DEFER_FINISH       - Execute deferred calls
```

### 6.2.9. Concurrency

```
FORK               - Spawn new coroutine
```

## 6.3. AST Structure to Bytecode

*Code Generation Traversal*

```
AST Node                     Generated Instructions
─────────────────────────────────────────────────────────────────────────────────────
────────────────────
AstArithmeticExpression     gen(left)
    op: "+"                  gen(right)
    left: a                  ADD
    right: b

AstAssignment                gen(expression)
    name: "x"                STORE "x"
    expr: 42

AstIfStatement               gen(condition)
    condition                JUMP_IF_FALSE else_label
    ifBody                   gen(ifBody)
    elseBody                 JUMP end_label
                             LABEL else_label
                             gen(elseBody)
                             LABEL end_label
```

```
AstFunctionCall             gen(arg)   ; for each arg
    name: "foo"             CALL "foo", 2
    args: [a, b]
```

## 6.4. Example: Compiling a Simple Function

Source:

```
function int add(int a, int b) {
    return a + b;
}
```

Generated bytecode:

```
add:
    LOAD_I "a"        ; load value of a
    LOAD_I "b"        ; load value of b
    ADD               ; a + b
    RETURN            ; Return result
```

## 6.5. Instruction Stream

Instructions are stored in an `InstructionStream`:

```
struct InstructionStream {
    vector<Instruction*>* instructions;
    int startInsIndx;   // Entry point offset
    int lastInsIdx;     // Last instruction index
};

struct EntryPoint {
    int mainIp;                  // main() entry point
    InstructionStream* ins;
    CodeGenConstantPool* constantPool;
};
```

**umar**

Although omitted in the code example, code generation requires the type checker to have produced a `SymbolTable` to allow type resolution for a node. Code generation can become complex depending on the AstNode being evaluated but the core mechanism is the same for every node.

# Chapter 7. Code Transformation

**umar**

Code Transformation in noC has nothing to do with improved code generation. It is a mechanism for rewriting parts of the AST and substituting nodes. For example, once polymorphism is resolved for a function call, the AstFunctionCall node is replaced with the polymorphic expanded function call.

Code transformations are used heavily to support the following:

- match statements
- Polymorphic function expansion
- Non local identifiers
- Resolving lambda calls
- Creations of closures

**umar**

*AstNode*

```
struct AstNode {
    NodeType type;
    AstNode* sub; // This is a substition field that will be used to replace
};
```

Every `AstNode` has a `sub` field, short for substitution, that allows the node to be replaced by an entirely new node. The idea for this is from Jonathan Blow's programming language JAI. During code generation, if a node has a non-null substitution then that is used to generate the code.

## 7.1. Support match statement

*match statement on enums*

```
enum OperatorType {
    ADD,
    MINUS,
    MULTPLY,
    DIVIDE,
    NO_MATCH
}

operator_type = OperatorType.ADD
match operator_type {
    case MINUS : {
    }

    case ADD : {

    }

    case MULTPLY : {

    }

    case DIVIDE : {

    }

    case NO_MATCH : {

    }
}
```

**umar**

When the compiler sees a match statement it transforms the code to a chain of if else statements. The `AstMatchNode` is substituted into `AstIfStatement`: matchnode→sub = createIfStatement();

*The code is transformed into the following if else statments*

```
    if (type == MINUS) {

    }

    else if (type == ADD) {

    }

    else if (type == MULTPLY) {

    }

    else if (type == DIVIDE) {

    }

    else if (type == NO_MATCH) {

    }
}
```

**umar**

You can find the transformation code in `code_transformer.cpp`. I have left out the code because the implementation detail is not really interesting. The important aspect to understand about the code is the substitution field.

## 7.2. Supporting Non local identifiers

*Local functions in noC*

```
function outer() {
    x = 10; // depth = 1
    function inner() { // depth = 2
        print(x); // x accessed at depth 2, declared at depth 1
    }
    inner();
}
```

> **umar**
>
> One of my favourite features in a language is local functions; the ability to nest a function inside another function. However, a local function can reference a variable that is in the outer function's stack frame.
>
> Supporting non local identifiers is effectively calculating the stack difference when a variable is referenced.

### Phase 1: Detection (Type Checking)

1. Parser creates AST with plain `Identifier` nodes
2. Type checker traverses the AST
3. For each identifier, `searchSymbol()` walks the scope chain:
    - First searches all block scopes within the current function
    - Then follows `parent` pointer to outer functions
    - Finally checks global symbol table
4. When a symbol is found at a different `function_depth_level`, `markSymbolNonLocal()` creates a `NonLocalIdentifier`

### Phase 2: Transformation

After type checking, the `resolve_closures` pass processes each function:

1. For each `NonLocalIdentifier`
2. Calculate `call_stack_difference = function_depth - symbol_depth`
3. Create a new `ScopedIdentifier` with this difference

**NOTE** A scoped identifer is an `AstIdentifierNode` with an additonal field called the stack difference.

1. Attach it to the original node via the `sub` (substitution) field

**NOTE** The substitution mechanism allows transformation without modifying the original AST structure. The code generator checks for `node→sub` and uses the substituted node if present.

### Phase 3: Code Generation

When the code generator encounters a `ScopedIdentifier`:

```
emit(LOAD_I)
.symbol = scoped_identifier->name
.arg1 = scoped_identifier->call_stack_difference
```

The runtime uses `call_stack_difference` to walk up the correct number of activation frames to find the variable.

## Example

```
.Local function in noC
function outer() {
    x = 10 // depth = 1
    function inner() { // depth = 2
        print(x) // x accessed at depth 2, declared at depth 1
    }
    inner();
}
```

1. During type checking of `inner()`, the identifier `x` is looked up
2. `x` is found in `` outer□s scope with `function_depth_level = 1 ``
3. Current function (`inner`) has `functionDepthLevel = 2`
4. A `NonLocalIdentifier` is created
5. During transformation: `call_stack_difference = 2 - 1 = 1`
6. A `ScopedIdentifier(name="x", call_stack_difference=1)` replaces the identifier
7. At runtime, `LOAD_I` walks up 1 activation frame to find `x`

**NOTE** Recursive functions cannot reference non-local identifiers because the compiler cannot determine a fixed stack frame distance at compile time — the recursion depth varies at runtime.

# Part 3 : Booting Runtime

# Chapter 8. Virtual Machine and noCThreads

**umar**

A noCThread is a virtual thread that contains an instruction pointer (ip) and a stack. A virtual machine runs a noCThread by following the standard fetch, decode, execute cycle that a real CPU would do.

```
struct noCThread {

    int vThreadId;
    int Ip;
    stack<NoCRuntimeObjectContainer*>* operands;
    atomic<ThreadState>* threadState;
    ActivationFrame* currentActivationFrame;
    InstructionStream* instructionStream;

};
```

The noCThread contains that actual execution state actual execution state:

| Field | Purpose |
|---|---|
| vThreadId | Virtual thread identifier |
| Ip | Instruction pointer - current position in bytecode |
| operands | The operand stack for expression evaluation |
| threadState | Atomic state (Created, Ready, Running, Waiting, Parked, Finished, etc.) |
| currentActivationFrame | Top of the call stack |
| instructionStream | The bytecode being executed |

## Main Interpreter Loop

The function runThreadInner() is the heart of the VM:

```
void runThread(noCThreadCtx* thread_ctx) {
    while (isIpValid(thread_ctx->thread)) {
        Instruction* instruction = ... // fetch at IP
        switch (instruction->op_code) {
            case LOAD_I:    // load variable
            case STORE:     // store variable
            case ADD:       // arithmetic
            case CALL:      // function call
            // ... ~40 opcodes
        }
    }
}
```

# Chapter 9. Scheduler

**umar**

The noCThreads are run by the scheduler.

The scheduler is a work-stealing design where each worker has its own run queue. When a worker’s queue is empty, it steals tasks from other workers. Each Worker is tied to an OS thread and workers run Tasks. Tasks are just noCThreads and the VM interpreter is run for that noCThread.

When a noCThread needs to yield (I/O call, lock acquire, etc.) it is parked. The Worker is free to pull another Task from its run queue. This is in essence how runtime systems support virtual threads.

The scheduler code is incredibly simple: there is just one private queue per worker and all Tasks have equal FIFO priority. This will inevitably create situations of starvation where a Task is queued up behind less important Tasks.

For example, a background Task that is processing some metrics should have less priority than a Task that is ready to read bytes from a socket to respond to a user HTTP request.

The scheduler provides none of these complexities.

## Starting the Scheduler

```
void run_scheduler(Scheduler* scheduler, int num_workers, noCThread* mainThread) {

    // 1. Create N workers
    for (int i = 0; i < num_workers; i++) {
        Worker* worker = create_worker(i);
        scheduler->workers->push_back(worker);
        worker->victims = scheduler->workers;  // All workers visible
    }

    // 2. Start OS thread for each worker
    for (auto worker : *scheduler->workers) {
        pthread_create(worker->kernel_thread, NULL, worker_loop, worker);
    }

    // 3. Schedule main thread
    push_task(mainThread);

    // 4. Wait for completion
    for (auto worker : *scheduler->workers) {
        pthread_join(*worker->kernel_thread, NULL);
    }
}
```

## The Worker Loop

```
void worker_loop(Worker* worker) {
    // Set up jump point for parking
    setjmp(*worker->reset_continuation->buf);

    for (;;) {
        // 1. Try own queue first
        noCThread* thread = dequeue(worker->runQueue);

        // 2. If empty, try stealing
        if (thread == NULL) {
            thread = steal_task(worker);
        }

        // 3. Still nothing? Loop again
        if (thread == NULL) {
            continue;
        }

        // 4. Run the thread
        thread->tcb->threadState = Running;
        runThread(thread);  // runs main interpreter loop
    }
}
```

## Key Concepts

| Concept | Description |
|---|---|
| Worker | An OS thread that runs virtual threads |
| Run Queue | Each worker has its own task queue |
| Work Stealing | Idle workers steal from other workers' queues |
| Parking | Thread yields control back to worker via longjmp |
| Resuming | Thread is re-enqueued to continue execution |

---

*Scheduler Architecture*

```
              +-------------------+
              |    Scheduler      |
              +-------------------+
                       |
     +-----------------+-----------------+
     |                 |                 |
+--------+        +--------+        +--------+
| Worker |        | Worker |        | Worker |
|   0    |        |   1    |        |   2    |
+--------+        +--------+        +--------+
     |                 |                 |
+--------+        +--------+        +--------+
|RunQueue|        |RunQueue|        |RunQueue|
| T1, T4 |        | T2, T5 |        | T3     |
+--------+        +--------+        +--------+
     |                 |                 |
     +---------+-------+---------+-------+
               |                 |
          +--------+        +--------+
          |IO Mng  |        |GC Flags|
          |(shared)|        |(shared)|
          +--------+        +--------+
```

**umar**

During my study I could not find any detailed formal literature on scheduler designs. Video publications on youtube were the best references for ideas around this.

Many runtime systems follow the same design principles of work stealing, some runtimes amortize the cost by stealing a large bulk, some systems push work to a global shared queue if their own private queues have large number of tasks.

Other runtime systems have different run queues with different levels as Tasks are boosted or unboosted as they move between queues.

I also am not sure of any runtime schedulers that behave like an OS scheduler with consideration to NUMA architectures. In NUMA architectures the OS scheduler takes the motherboard’s topology into account, migrating threads to CPU cores whose memory latency will be better. I have not researched into whether any runtime scheduler does this.

I refrained from adding any of this complexity because I wanted to profile the cost of having the simplest scheduler possible. This complements my vision of this project to have a sandbox where I can run precise instrumentation and experimentation.

# Chapter 10. Runtime objects

All runtime values are represented as objects with a common header.

## 10.1. The Object Header

```
struct NoCRuntimeObjectContainer {
    RuntimeObjectType type;   // Discriminator
    int type_number;          // For struct identification
};

enum struct RuntimeObjectType {
    NDimensionVector,         // Arrays
    Int,                      // Integer primitive
    Bool,                     // Boolean primitive
    Double,                   // Double primitive
    Char,                     // Character primitive
    NativeStringStruct,       // String (struct wrapper)
    NonNativeStringWrapper,   // C++ string wrapper
    StructObjType,            // User-defined struct
    ExternalHandle,           // External resource
    Closure,                  // Closure object
    Null                      // Null singleton
};
```

## 10.2. Primitive Types

Primitives are boxed in simple containers:

```
struct NoCIntRuntimeObject {
    NoCRuntimeObjectContainer header;
    int value;
};

struct NoCBoolRuntimeObject {
    NoCRuntimeObjectContainer header;
    bool value;
};

struct NoCDoubleRuntimeObject {
    NoCRuntimeObjectContainer header;
    double value;
};

struct NoCCharRuntimeObject {
    NoCRuntimeObjectContainer header;
    char value;
};
```

*Primitive Object Layout*

```
NoCIntRuntimeObject (12 bytes total):
+----------------------+
| header.type = Int     |  4 bytes
| header.type_number   |  4 bytes
+----------------------+
| value = 42           |  4 bytes
+----------------------+
```

## 10.3. Vectors

Arrays are represented as vectors with slice information:

```
typedef char no_c_field;

struct RuntimeDimension {
    int size;
    RuntimeDimension* next;   // For multi-dimensional
};

struct Slice {
    int num_dimensions;
    int startingOffset;
    RuntimeDimension* dimension;
};

struct NoCVectorRuntimeObject {
    NoCRuntimeObjectContainer header;
    size_t len;               // Total element count
    Slice slice;              // Dimension info
    no_c_field** start_of_vec;      // Pointer to elements
};
```

*2D Array Layout*

```
NoCVectorRuntimeObject for int[3][2]:
+------------------+
| header           |
+------------------+
| len = 6          |
+------------------+
| slice:           |
|   num_dims = 2   |
|   offset = 0     |
|   dimension -----+---> RuntimeDimension(3)
```

```
+------------------+              |
| start_of_vec ----+--+           v
+------------------+  |    RuntimeDimension(2)
                      |
                      v
              +---+---+---+---+---+---+
              | 0 | 1 | 2 | 3 | 4 | 5 |
              +---+---+---+---+---+---+
              [0,0][0,1][1,0][1,1][2,0][2,1]
```

## 10.4. User-Defined Structs

Structs use a flexible array member for fields:

```
struct NoCStructContainer {
    NoCRuntimeObjectContainer header;
    no_c_field* fields[0];   // Flexible array - each element is a pointer
};
```

*Struct Layout Example*

```
struct Person { string name; int age; }

NoCStructContainer:
+------------------+
| header.type =    |
|   StructObjType  |
| header.type_num  |
|   = Person_ID    |
+------------------+
| fields[0] -------+--> NoCNativeString("Alice")
+------------------+
| fields[1] -------+--> NoCIntRuntimeObject(30)
+------------------+
```

## 10.5. Memory Layout Summary

| Type | Size | Layout |
|---|---|---|
| Int | 12 bytes | header + 4-byte value |
| Bool | 9 bytes | header + 1-byte value |
| Double | 16 bytes | header + 8-byte value |
| Char | 9 bytes | header + 1-byte value |
| Struct | 8 + (8 * fields) | header + pointer array |
| Vector | 32 + elements | header + slice + data pointer |

# 10.6. Memory Layout Summary

```
Primitive (e.g., Int):
+----------------------+-----------+
| header (type=Int)    | value     |
+----------------------+-----------+

Vector:
+----------------------+-----+-------+-------------------------+
| header (type=Vector) | len | slice | start_of_vec[0..len-1]  |
+----------------------+-----+-------+-------------------------+
                                           |
                                           v
                              +-----+-----+-----+-----+
                              | ptr | ptr | ptr | ... |
                              +--+--+--+--+--+--+-----+
                                 |     |     |
                                 v     v     v
                              [elem] [elem] [elem]

Struct:
+------------------------------+----------------------+
| header (type=Struct, type_#) | fields[0..n-1]       |
+------------------------------+----------------------+
                                      |
                        +-------------+-------------+
                        v             v             v
                   [field 0]     [field 1]     [field 2]
```

# Chapter 11. Function calls and Activation Frames

When a function is called, the VM creates an **activation frame** to hold the function's execution context. Each frame stores local variables, tracks where to return, and links to other frames for scope resolution and stack unwinding.

## 11.1. Activation Frame Structure

The activation frame is defined in `include/noCThread/noCThread.h`:

```
struct ActivationFrame {
    map<string, NoCRuntimeObjectContainer*>* storage;  // Local variables
    vector<Instruction*>* deferredInstructions;        // Defer cleanup

    FunctionSymbol* functionSymbol;      // Function metadata
    ActivationFrame* previousFrame;      // Dynamic link (caller)
    ActivationFrame* staticLink;         // Lexical link (enclosing scope)
    int returnAddress;                   // Where to resume after return
    int throwReturnAddress;              // Exception handler (-1 if none)
    noCThread* owningThread;             // Owning thread
};
```

## 11.2. Creating a New Frame

When a function is called, `createNewActivationFrame()` in `src/vm/noC_vm.cpp` sets up the frame:

```
ActivationFrame* createNewActivationFrame(noCThread* thread,
                                          ActivationFrame* staticLink,
                                          int returnAddress,
                                          int throwAddress) {
    ActivationFrame* newFrame = new ActivationFrame();
    newFrame->storage = new map<string, NoCRuntimeObjectContainer*>();
    newFrame->returnAddress = returnAddress;
    newFrame->throwReturnAddress = throwAddress;

    // Link to caller's frame (dynamic link)
    newFrame->previousFrame = thread->tcb->currentActivationFrame;

    // Link to enclosing scope (static link)
    newFrame->staticLink = staticLink;

    // Push frame onto the stack
    thread->tcb->currentActivationFrame = newFrame;
    return newFrame;
}
```

## 11.3. The CALL Instruction

The `CALL` instruction in `src/vm/noC_vm.cpp` initiates the call:

```
case Opcode::CALL:
{
    int returnAddress = getCurrentIp(thread_ctx) + 1;
    createNewActivationFrame(thread_ctx->thread,
                             defaultStaticLink(thread_ctx->thread),
                             returnAddress,
                             -1);  // No exception handler
    update_ip(thread_ctx, instruction->arg1);  // Jump to function
    break;
}
```

## 11.4. Binding Parameters at Function Entry

At the function's `LABEL` instruction, arguments are popped from the operand stack and stored in the frame:

```
case Opcode::LABEL:
{
    auto frame = thread_ctx->thread->tcb->currentActivationFrame;
    auto functionSymbol = frame->functionSymbol;

    // Pop arguments and store by parameter name
    for (size_t i = 0; i < functionSymbol->numberOfArgs; i++) {
        auto operand = pop_operand(thread_ctx);
        auto param = functionSymbol->params->at(i);
        frame->storage->insert_or_assign(param->name, operand);
    }
    increment_ip(thread_ctx);
    break;
}
```

## 11.5. The RETURN Instruction

When the function returns, `handleReturn()` in `src/vm/noC_vm.cpp` tears down the frame:

```
void handleReturn(noCThreadCtx* thread_ctx) {
    auto returnValue = pop_operand(thread_ctx);

    // Execute deferred instructions
    runDeferred(thread_ctx, thread_ctx->thread->tcb->currentActivationFrame);

    // Push return value for caller
    push_operand(thread_ctx, returnValue);

    // Restore instruction pointer
    thread_ctx->thread->tcb->Ip =
        thread_ctx->thread->tcb->currentActivationFrame->returnAddress;

    // Pop frame (restore caller's frame)
    thread_ctx->thread->tcb->currentActivationFrame =
        thread_ctx->thread->tcb->currentActivationFrame->previousFrame;
}
```

## 11.6. Dynamic Link vs Static Link

The two frame links serve different purposes:

**Dynamic link (previousFrame)** points to the **caller's** frame. It forms the runtime call chain and is used for returning and exception unwinding.

**Static link (staticLink)** points to the **lexically enclosing** scope's frame. It enables closures to access variables from outer scopes. For regular calls, static link equals previous frame. For closure calls, it points to the captured frame.

## 11.7. Exception Handling and Stack Unwinding

Exception handling uses the `throwReturnAddress` field in activation frames. When a function is called with `try`, the VM sets this field to mark where control should return if an exception is thrown.

### 11.7.1. CALL vs CALL_WITH_TRY_CATCH

A regular `CALL` sets `throwReturnAddress` to -1, meaning no handler:

```
case Opcode::CALL:
{
    int returnAddress = getCurrentIp(thread_ctx) + 1;
    createNewActivationFrame(thread_ctx->thread,
                             defaultStaticLink(thread_ctx->thread),
                             returnAddress,
                             -1);  // No exception handler
    update_ip(thread_ctx, instruction->arg1);
    break;
}
```

A `CALL_WITH_TRY_CATCH` sets `throwReturnAddress` to the instruction after the call:

```
case Opcode::CALL_WITH_TRY_CATCH:
{
    int returnAddress = getCurrentIp(thread_ctx) + 1;
    createNewActivationFrame(thread_ctx->thread,
                             defaultStaticLink(thread_ctx->thread),
                             returnAddress,
                             returnAddress);  // Handler points here
    update_ip(thread_ctx, instruction->arg1);
    break;
}
```

### 11.7.2. Stack Unwinding

When a `THROW` instruction executes, `throw_error_internal()` walks up the call stack looking for a frame that can handle the exception:

```
void throw_error_internal(noCThreadCtx* thread_ctx, NoCStructContainer* error) {
    ActivationFrame* current = thread_ctx->thread->tcb->currentActivationFrame;

    // Walk up the call stack
    while (current != NULL) {
        // Run deferred cleanup for each frame we unwind
        runDeferred(thread_ctx, current);

        // Check if this frame has an exception handler
        if (current->throwReturnAddress != -1) {
            // Found a handler - restore to caller's frame
            thread_ctx->thread->tcb->currentActivationFrame = current->previousFrame;
            update_ip(thread_ctx, current->throwReturnAddress);

            // Push error for the catch code to inspect
            push_operand(thread_ctx, error);
            push_operand(thread_ctx, &nullObject);
            return;
        }

        current = current->previousFrame;
    }

    // No handler found - unhandled exception
    throw NoCUnhandledError(error);
}
```

The unwinding process:

1. Starts at the current frame where the exception was thrown
2. Walks backward through `previousFrame` links (the dynamic call chain)

3. Executes deferred instructions at each frame for cleanup
4. Stops when it finds a frame with `throwReturnAddress != -1`
5. Restores execution to that handler location

# 11.8. Defer Statements

The `defer` keyword schedules a function call to execute when the current function exits, regardless of whether it exits normally via return or abnormally via exception. This provides a clean way to handle resource cleanup.

## 11.8.1. Syntax

```
defer functionCall(args);
```

The deferred function must:

- Return `void` - since the result cannot be used
- Not throw exceptions - deferred calls run during cleanup when throwing would be problematic

The type checker enforces these constraints:

```
if (statement->type == Defer) {
    AstDefer* deferStatement = (AstDefer*)statement;
    typeCheckStatement(symbolTable, deferStatement->defferedFunctionCall, ...);
    FunctionSymbol* deferredFunctionSymbol = getFunctionSymbol(symbolTable,
                                                                deferStatement-
>defferedFunctionCall->name);

    if (deferredFunctionSymbol->markedAsThrows) {
        emit_error_info(deferStatement->defferedFunctionCall,
                        "deferred function call cannot throw");
    }

    if (!isReturnTypeVoid(deferredFunctionSymbol->returnType)) {
        emit_error_info(deferStatement->defferedFunctionCall,
                        "deferred function call should be void");
    }
}
```

## 11.8.2. Example Usage

A common use case is mutex unlocking:

```
function doWork(mutex mutex, BoxedInt counter) {
    lock(mutex);
    defer unlock(mutex);  // Will run when function exits
    for (i=0; i<10000; i++) {
        add(counter, 1);
    }
}  // unlock(mutex) executes here automatically
```

The `defer` ensures the mutex is released even if code between `lock` and the function exit throws an exception.

### 11.8.3. Code Generation

When the compiler encounters a `defer` statement, it generates:

```
DEFER start_addr, end_addr    ; Register deferred call, skip to end
; ... deferred function call instructions ...
DEFER_FINISH                  ; Marks end of deferred block
```

```
else if (node->type == Defer) {
    AstDefer* astDefer = (AstDefer*)node;
    Instruction* deferins = emit(stream, Opcode::DEFER);
    deferins->arg1 = getCurrentOff(stream);  // Start of deferred code
    gen_internal(stream, labelMap, symbolTable, astDefer->defferedFunctionCall, ...);
    emit(stream, Opcode::DEFER_FINISH);
    deferins->arg2 = getCurrentOff(stream);  // End of deferred code
}
```

The key insight is that the deferred instructions are emitted inline but skipped during normal execution. The `DEFER` instruction records where the deferred code lives, then jumps past it.

### 11.8.4. Runtime Execution

#### The DEFER Opcode

When the VM executes `DEFER`:

```
case Opcode::DEFER:
{
    // Record this instruction for later execution
    addDeferedInstruction(thread_ctx, instruction);
    // Skip past the deferred code
    update_ip(thread_ctx, instruction->arg2);
    break;
}
```

The instruction is stored in the current activation frame's `deferredInstructions` list:

```
void addDeferedInstruction(noCThreadCtx* thread_ctx, Instruction* deferred) {
    thread_ctx->thread->tcb->currentActivationFrame->deferredInstructions->push_back
(deferred);
}
```

### Activation Frame Structure

Each activation frame maintains a list of deferred instructions:

```
ActivationFrame* createNewActivationFrame(noCThread* thread,
                                          ActivationFrame* staticLink,
                                          int returnAddress,
                                          int throwAddress) {
    ActivationFrame* newFrame = new ActivationFrame();
    newFrame->deferredInstructions = new vector<Instruction*>();  // Defer list
    newFrame->returnAddress = returnAddress;
    newFrame->throwReturnAddress = throwAddress;
    // ...
}
```

### Running Deferred Instructions

When a function returns (either normally or during unwinding), deferred instructions execute:

```
void runDeffered(noCThreadCtx* thread_ctx, ActivationFrame* frame) {
    // Restore the frame context
    thread_ctx->thread->tcb->currentActivationFrame = frame;

    // Execute each deferred instruction
    for (auto deferredInstruction : *frame->deferredInstructions) {
        thread_ctx->thread->tcb->Ip = deferredInstruction->arg1;  // Jump to deferred
code
        runThreadInner(thread_ctx);  // Execute until DEFER_FINISH
    }
}
```

The `DEFER_FINISH` opcode simply returns from `runThreadInner`, allowing the next deferred instruction to execute:

```
case Opcode::DEFER_FINISH:
{
    return;  // Exit interpreter loop for this deferred block
}
```

### Integration with Return

Both normal and void returns run deferred instructions before exiting:

```
void handleReturn(noCThreadCtx* thread_ctx) {
    auto currentReturnValue = pop_operand(thread_ctx);
    runDeffered(thread_ctx, thread_ctx->thread->tcb->currentActivationFrame);
    // ... restore frame and push return value ...
}

void handleReturnFromVoid(noCThreadCtx* thread_ctx) {
    runDeffered(thread_ctx, thread_ctx->thread->tcb->currentActivationFrame);
    // ... restore frame ...
}
```

### Integration with Exception Unwinding

During stack unwinding, each frame's deferred instructions run before moving to the next frame:

```
void throw_error_internal(noCThreadCtx* thread_ctx, NoCStructContainer* error) {
    ActivationFrame* current = thread_ctx->thread->tcb->currentActivationFrame;

    while (current != NULL) {
        // Run deferred cleanup for each frame we unwind
        runDeferred(thread_ctx, current);

        if (current->throwReturnAddress != -1) {
            // Found handler, restore and continue
            // ...
            return;
        }
        current = current->previousFrame;
    }
    // No handler found
    throw NoCUnhandledError(error);
}
```

## 11.8.5. Execution Order

Deferred calls execute in LIFO (last-in, first-out) order, since they are appended to a list and executed in forward order of registration. Multiple defers in a function:

```
function example() {
    defer cleanup1();
    defer cleanup2();
    defer cleanup3();
    // ...
}
```

```
// Execution order: cleanup1(), cleanup2(), cleanup3()
```

This matches the order in which resources are typically acquired and should be released.

# Chapter 12. Representing Closures

Closures capture their lexical environment, allowing functions to access variables from their defining scope even after that scope has exited.

Closures is the basis for running async no routines.

## 12.1. Closure Object Structure

```
// Closures are structs with special fields
#define CLOSURE_FP_FIELD_NAME "fp"           // Function pointer (index)
#define CLOSURE_STATIC_LINK_FIELD_NAME "static_link"  // Captured frame

// A closure is a NoCStructContainer with type = Closure
// fields[0] = function index (int)
// fields[1] = static link (ActivationFrame*)
```

*Closure Memory Layout*

```
NoCStructContainer (Closure):
+----------------------+
| header.type = Closure|
| header.type_num      |
+----------------------+
| fields[0]: fp -------+--> NoCIntRuntimeObject(func_index)
+----------------------+
| fields[1]: static ---+--> ActivationFrame (captured)
|            link      |
+----------------------+
```

## 12.2. How Closure Creation Works

### 12.2.1. Identifying Escaping Functions

During type checking, functions that escape their scope are detected:

```
struct EscapingFunctionSymbol {
    AstNode* escapingNode;
    FunctionSymbol* escapingFunctionSymbol;
};

// Stored in SymbolTable
vector<EscapingFunctionSymbol*>* functionDefThatEscape;
```

A function escapes when: - It’s returned from another function - It’s stored in a variable that outlives its scope - It's passed to another function

### 12.2.2. AST Transformation

When a function escapes, its reference is transformed to closure creation:

```
// Before transformation
return innerFunc;

// After transformation (AST node substitution)
AstClosureCreation* closure = new AstClosureCreation();
closure->escapingFunctionName = "innerFunc";
originalNode->sub = closure;  // Substitute in AST
```

### 12.2.3. Code Generation

Closure creation generates these instructions:

```
; Create closure for function "innerFunc"
LOAD_IMMEDIATE func_index    ; Function's instruction offset
NEW closure_struct           ; Allocate closure struct
STORE_FIELD 0               ; Store fp
LOAD_FRAME                  ; Get current frame for static link
STORE_FIELD 1               ; Store static link
```

### 12.2.4. Runtime Closure Creation

```
NoCStructContainer* createClosureObject(Heap* heap,
                                        StructLayout* closureLayout) {
    NoCStructContainer* closure = createStructRuntimeObject(heap,
                                                            closureLayout);
    closure->header.type = RuntimeObjectType::Closure;
    return closure;
}
```

## 12.3. Calling a Closure

When calling a closure (CALL_LAMBDA):

```
case Opcode::CALL_LAMBDA: {
    // Pop closure from stack
    NoCStructContainer* closure =
        (NoCStructContainer*)tcb->operands->top();
    tcb->operands->pop();
    // Extract function pointer and static link
    NoCIntRuntimeObject* fpObj =
        (NoCIntRuntimeObject*)closure->fields[0];
    int funcIndex = fpObj->value;

    ActivationFrame* capturedFrame =
        (ActivationFrame*)closure->fields[1];

    // Create new frame with captured frame as static link
    ActivationFrame* newFrame = createNewActivationFrame(
        thread,
        capturedFrame,  // Static link to captured environment
        tcb->Ip + 1,
        -1
    );

    // Jump to function code
    tcb->currentActivationFrame = newFrame;
    tcb->Ip = funcIndex;
    break;
}
```

## 12.4. Variable Access via Static Links

When accessing a non-local variable inside a closure:

```
// x is 2 frames up the static chain
case Opcode::LOAD_SCOPED: {
    int levels = currentIns->arg1;
    string varName = currentIns->symbol;

    // Walk static links
    ActivationFrame* frame = tcb->currentActivationFrame;
    for (int i = 0; i < levels; i++) {
        frame = frame->staticLink;
    }

    // Load variable from target frame
    auto value = frame->storage->at(varName);
    tcb->operands->push(value);
    tcb->Ip++;
    break;
}
```

## 12.5. Summary Flow

```
1. Type Checker: Detect function escape
```

```
         |
         v
2. Mark function as needing closure
         |
         v
3. Transform AST: replace function ref with ClosureCreation
         |
         v
4. Code Gen: emit closure allocation + field stores
         |
         v
5. Runtime: closure object holds fp + static link
         |
         v
6. Call: CALL_LAMBDA uses closure's static link
```

## 12.6. Example: Closure Visualization

```
typedef function add_func :: (int) -> int;

function add_func makeAdder(int x) {
  function int add(int y)  {
     return x + y; // x captured from outer scope
  }
  return add; // 'add' escapes
}

function main() throws {
    adder = makeAdder(5); // Closure created
    result = adder(3); // Returns 8
    printf("The result is %d\n", result);
}
```

*Closure State After makeAdder(5) Returns*

```
adder5 (closure):
+---------------+
| fp: add_index |
+---------------+
| static_link --+---> makeAdder's Frame (kept alive!)
+---------------+          |
                           v
                     +------------+
                     | x: 5       |
                     +------------+

When calling adder5(3):
1. New frame created for 'add'
```

```
2. Static link set to captured frame (x=5)
3. 'add' can access x through static link
```

# Chapter 13. Forking no Routines and awaiting

noRoutines are lightweight threads that execute concurrently with the parent thread. When a noRoutine is forked, the compiler creates a closure around the anonymous routine, capturing the parent's activation frame. This allows the forked thread to access variables from the enclosing scope.

*A no routine is created by running an `no` before an anonymous function*

```
no function() {
    printf("This is running in another thread!");
}
```

## 13.1. The Forking Mechanism

When the compiler encounters a `no function() { … }` block, it treats the body as an anonymous function that "escapes" its defining scope. Since this function will execute on a separate thread after the fork, it needs access to the parent's environment.

The solution reuses the closure mechanism:

1. The anonymous function is marked as escaping during type checking
2. A closure is created that packages the function pointer with the current activation frame
3. At fork time, the closure is passed to the new thread
4. The forked thread calls the closure, which sets up the static link back to the parent's frame

## 13.2. Parameter Capture

Variables referenced inside the noRoutine body are captured at fork time. The `FORK` instruction copies these values from the parent's scope into the forked thread's local storage. This ensures the forked thread has its own copies of the captured values.

## 13.3. Why Closures?

Using closures for noRoutines provides a unified model for capturing lexical scope. Whether a function escapes by being returned or by being forked onto another thread, the same closure mechanism ensures access to the enclosing environment through the static link.

## 13.4. VM Execution of FORK

When the VM executes a `FORK` instruction, it performs the following steps:

1. **Create the forked thread** — A new `noCThread` is created as a child of the parent. The VM first

checks if there’s a cached "dead" thread that can be reused; otherwise it allocates a new one.

2. **Copy the instruction stream** — The forked thread receives a copy of the instructions from the fork point to the end of the noRoutine body.
3. **Transfer the closure** — The closure (containing the function pointer and static link) is popped from the parent’s stack and stored in the forked thread under a special name.
4. **Capture parameters** — Each captured variable is loaded from the parent’s scope and stored in the forked thread’s local storage.
5. **Schedule the thread** — The forked thread is added to a worker’s task queue for execution.
6. **Return a future** — The parent thread receives a future handle that can be used to await the forked thread’s completion.

*src/vm/noC_vm.cpp - FORK instruction*

```
case Opcode::FORK:
{
    // 1. Create the forked thread
    auto forked_thread = createThreadFromParent(thread_ctx->thread, newInsStream);
    set_up_first_activation_frame(forked_thread);

    // 2. Transfer the closure
    auto noRoutineClosure = pop_operand(thread_ctx);
    storeSymbolInternal(forked_thread, "<noRoutine>", noRoutineClosure);

    // 3. Capture parameters from parent's scope
    for (int i = 1; i <= num_symbols_to_capture; i++) {
        Instruction* load_ins = getInstructionAt(thread_ctx, current_ip + i);
        auto captured_value = loadSymbol(thread_ctx, load_ins->symbol, load_ins->
arg1);
        storeSymbolInternal(forked_thread, load_ins->symbol, captured_value);
    }

    // 4. Schedule the thread
    schedule_new_thread(thread_ctx->scheduler, forked_thread);

    // 5. Return a future
    auto future = createExternalHandler(getHeap(thread_ctx->thread), forked_thread,
ExternalHandleType::Future);
    push_operand(thread_ctx, (NoCRuntimeObjectContainer*)future);
    break;
}
```

## 13.5. Scheduling and Worker Queues

The scheduler uses a pool of workers, each backed by an OS kernel thread. Each worker maintains a concurrent run queue of noRoutines ready to execute.

When a thread is forked, it is added to the **same worker’s queue** as its parent. This thread affinity preserves cache locality and reduces synchronization overhead.

The worker loop continuously:

1. Dequeues a thread from its own run queue
2. If empty, attempts to **steal** work from other workers' queues
3. If still no work, checks for completed I/O events
4. Executes the thread until it yields, blocks, or completes

This work-stealing design ensures that idle workers can help busy ones, balancing load across the thread pool while maintaining good locality for related threads.

# 13.6. Futures and Awaiting

When a noRoutine is forked, the parent receives a **future** — a handle that represents the eventual result of the forked computation. The parent can continue executing other work and later **await** the future to retrieve the result.

## 13.6.1. The Future Handle

A future is simply an external handle wrapping a pointer to the forked thread. This lightweight representation allows the parent to hold a reference to the forked computation without directly managing the thread.

Each thread has an associated `await_handle` structure containing:

- A **wait queue** of threads waiting for this thread to complete
- A **spin lock** protecting concurrent access
- Slots for the **result** and **error** values

## 13.6.2. Awaiting a Future

When the parent executes an `AWAIT` instruction, the `await()` function in `src/scheduler/scheduler_lock.cpp` handles the blocking:

```
void await(noCThread* src, noCThread* awaitOn) {
    await_handle_t* handle = awaitOn->tcb->await_handle;

    // Fast path: already completed
    if (threadStateAtLeast(awaitOn, Signaled)) {
        return;
    }

    // Add ourselves to the wait queue
    ACQUIRE_SPIN_LOCK(handle->spin_lock);
    handle->wait_queue->push(src);
    handle->num_waiters++;
    RELEASE_SPIN_LOCK(handle->spin_lock);

    // Yield control back to the scheduler
    park_thread(src);
}
```

The `park_thread()` call uses `longjmp` to return control to the scheduler loop, allowing other threads to run while the parent waits.

### 13.6.3. Signaling Completion

When the forked thread finishes, `signal_waiters()` in `src/scheduler/scheduler_lock.cpp` notifies all waiting threads:

```
void signal_waiters(noCThread* src, NoCRuntimeObjectContainer* result,
                    NoCRuntimeObjectContainer* err) {
    await_handle_t* handle = src->tcb->await_handle;

    ACQUIRE_SPIN_LOCK(handle->spin_lock);

    // Store the result
    handle->result = result;
    handle->error = err;
    update_thread_state(src, Signaled);

    // Wake up all waiting threads
    while (handle->num_waiters > 0) {
        noCThread* waiter = handle->wait_queue->front();
        handle->wait_queue->pop();
        handle->num_waiters--;
        resume_thread(waiter);
    }

    RELEASE_SPIN_LOCK(handle->spin_lock);
}
```

Each `resume_thread()` call pushes the waiting parent back onto the scheduler's run queue, where it will eventually be picked up and continue execution.

### 13.6.4. Resuming the Parent

When the scheduler picks up the resumed parent thread:

1. Execution continues from where it was parked (at the `AWAIT` instruction)
2. The await function returns immediately (fast path — thread is now `Signaled`)
3. The result is retrieved from the `await_handle` and pushed onto the parent's operand stack
4. The parent continues normal execution

# Chapter 14. Locks: Reentrant Thread Synchronization

The `sleep_lock` is a synchronization primitive that supports reentrant locking for noC's virtual threads. Unlike traditional mutexes that would deadlock if the same thread tried to acquire a lock it already holds, the sleep_lock tracks ownership and allows the owning thread to proceed without blocking.

## 14.1. The sleep_lock Structure

```
struct sleep_lock {
    atomic<int>* mutex;       // Holds MUTEX_FREE (0) or the owning thread's vThreadId
    atomic<int>* spin_lock;   // Protects wait_queue access
    queue<noCThread*>* wait_queue;  // Threads waiting for the lock
};
```

The key insight is that `mutex` does not simply store `MUTEX_FREE` (0) or `MUTEX_LOCKED` (1). Instead, when locked, it stores the **thread ID** (`vThreadId`) of the owning thread. This enables reentrant detection.

## 14.2. How Reentrant Locking Works

When a thread attempts to acquire the lock, the first check is whether the thread already owns it:

```
void sleep_lock_lock(sleep_lock* lock, noCThread* acquiring_thread) {
    int mutex_free = MUTEX_FREE;
    int mutex_locked = acquiring_thread->tcb->vThreadId;

    // Reentrant check: This thread already has the lock
    if (lock->mutex->load() == mutex_locked) {
        return;  // No blocking, proceed immediately
    }

    // Try to acquire the lock atomically
    if (lock->mutex->compare_exchange_strong(mutex_free, mutex_locked)) {
        return;  // Lock acquired
    }

    // Lock is held by another thread...
}
```

This pattern supports scenarios like:

```
function processData(mutex m) {
    lock(mutex);
    helperFunction(m)   // Calls another function that also needs the lock
    unlock(m);
}

function helperFunction(mutex m) {
    lock(m);
    // ... do work ...
}
```

Without reentrant support, `helperFunction()` would deadlock waiting for a lock that `processData()` already holds.

## 14.3. The Lock Acquisition Flow

*Lock Acquisition Sequence*

```
Thread attempts to acquire lock
            |
            v
+---------------------------+
| Is mutex == my vThreadId? |--YES--> Return (reentrant)
+---------------------------+
            | NO
            v
+---------------------------+
| CAS(mutex, FREE, myId)?   |--YES--> Return (acquired)
+---------------------------+
            | NO (another thread holds lock)
            v
+---------------------------+
| Acquire spin_lock         |
+---------------------------+
            |
            v
+---------------------------+
| CAS(mutex, FREE, myId)?   |--YES--> Release spin_lock, Return
+---------------------------+         (optimization: lock freed while waiting)
            | NO
            v
+---------------------------+
| Add thread to wait_queue  |
| Set thread state: Waiting |
| Release spin_lock         |
| Park thread (yield CPU)   |
+---------------------------+
```

The double-check after acquiring the spin_lock is an optimization: another thread may have released the lock while we were acquiring the spin_lock.

## 14.4. The Spin Lock vs Sleep Lock Distinction

The `sleep_lock` uses two synchronization mechanisms:

| Mechanism | Purpose |
|---|---|
| `mutex` (atomic CAS) | The main lock that protects the critical section. Non-blocking acquisition via compare-and-swap. |
| `spin_lock` (busy-wait) | Protects access to the `wait_queue`. Held only briefly to add/remove threads from the queue. |

```
#define ACQUIRE_SPIN_LOCK(lock)                      \
    for (;;) {                                       \
        int spin_lock_free = lock->load();           \
        while (spin_lock_free != MUTEX_FREE) { }     \
        int mutex_free = MUTEX_FREE;                 \
        int mutex_locked = MUTEX_LOCKED;             \
        if (lock->compare_exchange_strong(           \
                mutex_free, mutex_locked)) {         \
            break;                                   \
        }                                            \
    }
```

The spin_lock is appropriate here because it is held for an extremely short duration (just to push/pop from the queue). Threads do not park while holding the spin_lock.

## 14.5. Lock Release and Thread Wake-up

```
void sleep_lock_unlock(sleep_lock* lock, noCThread* owning_thread) {
    int mutex_locked = owning_thread->tcb->vThreadId;

    // Verify ownership
    if (lock->mutex->load() != mutex_locked) {
        throw runtime_error("unlocking a thread that is not owned");
    }

    // Release the main lock
    lock->mutex->store(MUTEX_FREE);

    // Check for waiting threads
    ACQUIRE_SPIN_LOCK(lock->spin_lock);
    if (lock->wait_queue->empty()) {
        RELEASE_SPIN_LOCK(lock->spin_lock);
        return;
    }

    // Wake up one waiting thread
    noCThread* thread = lock->wait_queue->front();
    lock->wait_queue->pop();
    resume_thread(thread);  // Re-enqueue to scheduler
    RELEASE_SPIN_LOCK(lock->spin_lock);
}
```

The unlock sequence:

1. Verify the calling thread owns the lock
2. Release the mutex (store `MUTEX_FREE`)
3. Acquire spin_lock to safely access wait_queue
4. If threads are waiting, wake up the first one via `resume_thread()`
5. Release spin_lock

## 14.6. Visualization: Sleep Lock State Machine

```
                    LOCK STATES
                    ===========


    +-------------+                      +------------------+
    |  MUTEX_FREE |                      | MUTEX = ThreadID |
    |    (0)      |                      |    (locked)      |
    +-------------+                      +------------------+
          |                                      |
          | Thread A calls                       | Thread A calls
          | lock(A)                              | unlock(A)
          | CAS(0 -> A.id)                       | store(0)
          v                                      v
    +-------------+                      +------------------+
    | MUTEX = A   |--------------------->|    MUTEX_FREE    |
    | (A owns it) |    unlock            |      (0)         |
    +-------------+                      +------------------+
          |
          | Thread A calls lock(A) again
          | mutex == A.id? YES -> return immediately
          v
    +-------------+
    | REENTRANT   |
    | (no block)  |
    +-------------+


                    THREAD B WAITING
                    ================


    +-------------+      +-------------+      +-------------+
    | Thread B    | ---> | wait_queue  | ---> | Thread B    |
    | calls       |      | [B]         |      | resumed     |
    | lock(B)     |      | (parked)    |      | when A      |
    | CAS fails   |      |             |      | unlocks     |
    +-------------+      +-------------+      +-------------+
```

**umar**

The sleep_lock implementation is incredibly simple and does not handle fairness. Newly arriving threads that find the mutex free can acquire it immediately, even if other threads have been waiting longer. This can lead to starvation in high-contention scenarios.

If you look at the go source code for mutexes you will find it tries to detect this starvation.

# Chapter 15. Non blocking I/O with epoll

The VM uses Linux epoll for non-blocking I/O. When a thread attempts an I/O operation that would block, it parks itself and registers interest with epoll. The worker loop polls for completion and resumes threads when their I/O is ready.

## 15.1. The io_manager Structure

Each worker has an io_manager that wraps epoll and tracks blocked threads:

```
struct io_manager {
    map<int, io_event*>* event_map;               // fd → event state
    map<int, noCThread*>* thread_blocked_descriptor;   // fd → blocked thread
    int epoll_fd;                                 // epoll instance
};
```

## 15.2. Parking a Thread on I/O

When a thread blocks on I/O (e.g., reading from a socket), it saves its operand stack and parks:

```
void park_thread_on_fd(noCThread* thread, int fd,
                       vector<NoCRuntimeObjectContainer*>* operands_to_save) {
    auto event_mng = get_event_mng(thread);
    auto event = get_event(event_mng, fd);

    // Save operands for restoration when I/O completes
    for (auto operand : *operands_to_save) {
        event->restore_operand_context->push(operand);
    }

    // Register thread as blocked on this fd
    event_mng->thread_blocked_descriptor->insert_or_assign(fd, thread);

    // Yield control back to the scheduler
    park_thread(thread);
}

void park_thread(noCThread* thread) {
    update_thread_state_weak(thread, ThreadState_E::Parked);
    Worker* worker = getPrivateWorker(thread);
    longjmp(*worker->reset_continuation->buf, 0);
}
```

## 15.3. Non-Blocking I/O Attempt

The VM attempts non-blocking reads/writes. If the operation would block, the thread parks:

```
io_read_write_status io_on_socket(io_type type, int socket_fd,
                                  char* buffer, size_t len) {
    int bytes = (type == io_type::READ)
        ? read(socket_fd, buffer, len)
        : write(socket_fd, buffer, len);

    if (bytes >= 0) {
        return { bytes, io_success::GOOD };
    }
    if (errno == EAGAIN || errno == EWOULDBLOCK) {
        return { 0, io_success::WOULD_BLOCK };
    }
    return { 0, io_success::FAILED };
}
```

In the VM instruction handler:

```
// Attempt non-blocking read
io_read_write_status status = io_on_socket(io_type::READ, socket_fd, buffer, len);

if (status.status == io_success::WOULD_BLOCK) {
    // Save operands and park until socket is ready
    auto restore = vector<NoCRuntimeObjectContainer*>();
    restore.push_back(socket_operand);
    restore.push_back(num_bytes_operand);
    park_thread_on_fd(thread_ctx->thread, socket_fd, &restore);
    return;
}

// Success - process the data
```

## 15.4. The Worker Loop and I/O Polling

When no threads are ready to run, the worker polls epoll for completed I/O:

```
void worker_loop(Worker* worker) {
    setjmp(*worker->reset_continuation->buf);  // Park returns here

    for (;;) {
        noCThread* thread = getTask(worker);
        if (thread == NULL) {
            thread = steal_task(worker);
            if (thread == NULL) {
                // No work available - check for I/O completion
                run_io_event_loop(worker->io_mng);
                continue;
            }
        }
        runThread(thread);
    }
}
```

## 15.5. Resuming Threads on I/O Completion

The event loop polls epoll and resumes any threads whose I/O is ready:

```
void run_io_event_loop(io_manager* mng) {
    struct epoll_event events[MAX_EVENTS];
    int count = epoll_wait(mng->epoll_fd, events, MAX_EVENTS, 0);  // Non-blocking

    for (int i = 0; i < count; i++) {
        int fd = events[i].data.fd;
        io_event* event = get_event(mng, fd);

        if (events[i].events & (EPOLLIN | EPOLLOUT)) {
            event->status = event_status::READY;
        }

        if (is_thread_blocked_on_fd(mng, fd)) {
            resume_thread_from_io_event(mng, event);
        }
    }
}

void resume_thread_from_io_event(io_manager* mng, io_event* event) {
    noCThread* thread = event->waiting_thread;

    // Restore saved operands
    while (!event->restore_operand_context->empty()) {
        auto operand = event->restore_operand_context->top();
        event->restore_operand_context->pop();
        thread->tcb->operands->push(operand);
    }

    mng->thread_blocked_descriptor->erase(event->descriptor_fd);
    push_task(thread);  // Re-queue for execution
}
```

## 15.6. Flow Summary

```
1. Thread attempts read() on socket
2. Returns EWOULDBLOCK - socket not ready
3. Thread saves operands, registers with epoll, parks via longjmp

4. Worker loop has no tasks
5. Calls epoll_wait(timeout=0) - non-blocking poll
6. Socket becomes ready, epoll returns event

7. Worker finds blocked thread for that fd
8. Restores operands, pushes thread to run queue
9. Thread resumes, retries read() - now succeeds
```

This design allows many noC threads to perform concurrent I/O without blocking kernel threads. Threads park cooperatively and resume when their I/O completes.

**umar**

Non blocking I/O relies on OS support. Different OS's have different interfaces to intiate non blocking I/O. Linux has `epoll`, `mac` and `free bsd` have kqueue, Windows has I/O completion ports

The `Libuv` project provides one abstract interface that supports all the OS's. This is incredibly useful when creating a runtime for a langauge that will support many different OS's. You can take a depenancy on libuv and let libuv handle the implementation details.

Another fascniating interface is I/O uring in the linux kernel. It removes the overhead of querying I/O status with syscalls by having a shared queue between the user space and the kernel.

DPDK is antoher projects that removes the kernel networking stack and allows an application to directly read from the network card. I am not sure if DPDK and the kernel network stack can run at the same time.

Professionally I have not seen I/O uring or DPDK used in production, if you have experience with these please do let me know how you are using them.

# Chapter 16. External functions

External functions are native implementations callable from noC code. When an external function is called it calls into a C++ function that handles that external function

*External functions for socket*

```
external function socket create_socket() throws;
external function connect_socket(socket socket, string address, int port) throws;

// Read and write socket can both read/write partial data
external function byte[] read_socket(socket socket, int bytes_to_read) throws;
external function int write_socket_base(socket socket, byte[] data, int offset)
throws;
external function close_socket(socket socket);
```

External functions receive the calling thread and access the operand stack:

*Handles the external function socket accept*

```
else if (instruction->symbol == SOCKET_ACCEPT) {

        auto server_socket_operand = pop_external_operand(thread_ctx);
        int sock_fd = get_socket_fd(server_socket_operand);

        io_accept_status accept_status = accept_on_socket(sock_fd);

        if (accept_status.status == io_success::WOULD_BLOCK) {
            auto restore_operands = vector<NoCRuntimeObjectContainer*>();
            restore_operands.push_back((NoCRuntimeObjectContainer*
)server_socket_operand);
            park_thread_on_fd(thread_ctx->thread, sock_fd, &restore_operands);
            return; // will never reach
        }

        if (accept_status.status == io_success::FAILED) {
            throw_error_internal(thread_ctx, "socket accept failed");
            return;
        }

        assert(accept_status.status == io_success::SUCCESS);
        assert(accept_status.client_fd != -1);

        BoxedDescriptor* boxedFd = alloc_pinned_boxed_fd(getHeap(thread_ctx->thread),
accept_status.client_fd);
        auto handler = createExternalHandler(getHeap(thread_ctx->thread), boxedFd,
ExternalHandleType::Socket);
        unpin_object((void*)boxedFd);
        push_operand(thread_ctx, (NoCRuntimeObjectContainer*)handler);
```

```
    }
```

# Chapter 17. Garbage Collection

The VM implements a stop-the-world mark-sweep garbage collector with per-thread private heaps. Memory is acquired from the OS via `mmap`, and each thread allocates from its own heap without locking.

## 17.1. Acquiring Memory from the OS

Memory is allocated in pages using `mmap` with optional guard pages for overflow detection:

```
page_alloc_result alloc_page_block_from_OS(size_t request_size) {
    int num_pages = request_size / DEFAULT_PAGE_SIZE;
    size_t total_size = (num_pages * DEFAULT_PAGE_SIZE) + GUARD_PAGE_SIZE;

    char* page_mem = (char*)mmap(NULL, total_size,
                                  PROT_READ | PROT_WRITE,
                                  MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

    // Protect the guard page
    mprotect(page_mem + (num_pages * DEFAULT_PAGE_SIZE),
             DEFAULT_PAGE_SIZE, PROT_READ);

    return { page_mem, num_pages * DEFAULT_PAGE_SIZE };
}
```

## 17.2. Per-Thread Private Heaps

Each thread receives its own heap, eliminating lock contention on allocation:

```
struct Heap {
    Page root_page;          // First page (embedded in heap header)
    Page* tail_page;         // Last allocated page
    FreeList freeList;       // Free block list
    HeapStats stats;
};

struct HeapNode {
    Header header;           // Size, flags (in-use, marked, pinned)
    HeapNode* next;          // Free list links
    HeapNode* prev;
};
```

Benefits of private heaps:

- **No lock contention** — Allocation is completely lock-free
- **Cache locality** — Thread-local memory stays in cache

- **Simpler GC** — Only need to coordinate worker pauses, not every allocation

## 17.3. The Free List

Allocation uses a first-fit strategy on a doubly-linked free list:

```
void* noC_alloc(Heap* heap, size_t request_size) {
    HeapNode* node = find_first_free_node(&heap->freeList, request_size);
    if (node == NULL) {
        grow_heap(heap);
        node = find_first_free_node(&heap->freeList, request_size);
    }

    // Split node if enough space remains
    if (node->header.availableSize - request_size >= MIN_BLOCK_SIZE) {
        HeapNode* remainder = split_node(node, request_size);
        add_to_free_list(heap, remainder);
    }

    remove_from_free_list(&heap->freeList, node);
    node->header.meta_data |= INUSE_BIT_MASK;

    return (char*)node + sizeof(HeapNode);
}
```

After GC, adjacent free blocks are coalesced to reduce fragmentation.

## 17.4. Pausing Workers for GC

GC runs at safe points (before calls, forks, and loop back-edges). Workers coordinate using atomic flags and a condition variable:

```
struct GC_Flag {
    atomic<GC_State>* state;    // Collection or NotCollecting
    pthread_mutex_t* lock;
    pthread_cond_t* gc_cond;
};
```

The initiating thread requests all workers to pause:

```
void garbage_collect() {
    // Atomically claim the collector role
    if (!gc_flags->state->compare_exchange_strong(NotCollecting, Collection)) {
        // Another GC in progress - wait for it
        wait_for_gc_complete();
        return;
    }
    // Request all workers to pause
    for (auto worker : *scheduler->workers) {
        worker->gc_request_flag = true;
    }

    // Spin until all workers acknowledge
    while (!all_workers_paused()) { }

    // ... mark and sweep ...

    // Resume workers
    for (auto worker : *scheduler->workers) {
        worker->gc_request_flag = false;
    }
    gc_flags->state->store(NotCollecting);
    pthread_cond_broadcast(gc_flags->gc_cond);
}
```

Workers check the flag at safe points:

```
void pause_worker_for_gc(Worker* worker) {
    if (worker->gc_request_flag) {
        worker->is_paused_for_gc = true;
        pthread_mutex_lock(gc_flags->lock);
        while (gc_flags->state->load() == Collection) {
            pthread_cond_wait(gc_flags->gc_cond, gc_flags->lock);
        }
        pthread_mutex_unlock(gc_flags->lock);
    }
}
```

## 17.5. Mark Phase

Marking starts from roots (thread stacks) and traces all reachable objects:

```
void mark_thread_stack(noCThread* thread) {
    ActivationFrame* frame = thread->tcb->currentActivationFrame;
    while (frame != NULL) {
        set_marked_bit(frame->frameRefTracker);

        // Mark all local variables
        for (auto& var : *frame->storage) {
            mark(var.second, symbolTable);
        }
        frame = frame->previousFrame;
    }
}
```

```
void mark(NoCRuntimeObjectContainer* obj, SymbolTable* symbolTable) {
    if (obj == NULL || isMarked(obj)) return;
    set_marked_bit(obj);

    switch (obj->type) {
    case StructObjType:
        // Mark all fields
        for (auto field : *structLayout->fields) {
            mark(struct->fields[field->offset], symbolTable);
        }
        break;
    case Closure:
        // Mark static link chain (captured scopes)
        ActivationFrame* link = get_static_link(obj);
        while (link != NULL) {
            mark(link->frameRefTracker, symbolTable);
            for (auto& var : *link->storage) {
                mark(var.second, symbolTable);
            }
            link = link->staticLink;
        }
        break;
    case Vector:
        // Mark all elements
        for (size_t i = 0; i < length; i++) {
            mark(vector->elements[i], symbolTable);
        }
        break;
    }
}
```

## 17.6. Sweep Phase

Unmarked objects are freed; marked objects have their mark bit cleared:

```
void sweep(Heap* heap) {
    Page* page = &heap->root_page;
    while (page != NULL) {
        HeapNode* node = page->node_start;
        while (node != NULL) {
            if (is_in_use(node) && !is_marked(node) && !is_pinned(node)) {
                // Unreachable - free it
                noC_free(node);
            } else if (is_marked(node)) {
                // Reachable - clear mark for next GC
                clear_mark_bit(node);
            }
            node = next_node(node, page);
        }
        page = page->next_page;
    }

    // Coalesce adjacent free blocks
    merge_heap(heap);
}
```

## 17.7. Page Recycling

Empty pages are returned to a global pool protected by a lock:

```
static GlobalFreePageList* freePageList;

void free_heap_pages(Heap* heap) {
    ticket_lock_lock(freePageList->lock);
    Page* page = heap->root_page.next_page;
    while (page != NULL) {
        Page* next = page->next_page;
        add_to_free_page_list(freePageList, page);
        page = next;
    }
    ticket_lock_unlock(freePageList->lock);
}
```

When a heap needs to grow, it first checks the global pool before calling `mmap`.

# References and Future Work

**umar**

The sandbox is ready to be played with and here but all ideas about future work are all focused on runtime optimisation. The font end of the compiler is not that interesting unless I make the type checking parralel.

- Add full debugger support using cppdap.
- Improving garbage collector to support generational collection
- Performance profiling
  - Measuring memory blow up when 1000’s of no routines are scheduled; each no routine has a private heap
  - Creating lock level starvation experiments to create pathalogical deadlocks
  - Supporting non blocking I/O with I_O Uring and measuring throughput of I/O bound and CPU bound no routines
  - Measuring scheduler throughput on NUMA servers vs UMA servers ( I will buy some cheap dual socket servers for this test )
  - Measure the impact of false sharing between no routines ( since each heap is private we should not see any issues )
- Many more ideas

Working on things with deeper thought creates more questions; these questions would have never been confronted had I not invested time in each line of code.

By no means is noC a huge body of work but it does give me the sandbox to do more interesting work.

**umar**

There are a whole host of materials that I studied to create this compiler. I still believe formal publications are still relevant for deep study because the "context" and depth is already there between the pages. You do not have to prompt a book to learn more; you just read the next page.

I decided to study Compiler construction the classical way by reading the literature instead of prompting AI for ideas. The literature on compiler construction is dense, and I was satisfied with the progress I was making.

The only time I did use the AI was to go over material I studied for CPU cache consistency. All large language models have a learning mode and can compliment individual study of materials really well. I ended up not creating my own lock intrinsics, but AI in learning mode was a fantastic resource here to solidify my understanding.

> Moving forward I will compliment my study of formal materials with AI assited learning.

# Compiler Books

**Understanding and Writing Compilers: A do-it-yourself guide**
*Richard Bornat*

A book for the 70's and unlike many other compiler publications, Richard Bornat puts no emphasis on grammars and describes the simplicity of Tree Walking. Once you understand Tree Walking you can build a language. The expression evaluation code in the parser is from the algorithm Richard Bornat describes.

**Introduction to Compilers and Language Design, Second Edition**
*Prof. Douglas Thain*

Whereas Richard Bornat describes the book, Douglas Thain does provide implementation in C. A very gentle succinct read that also covers different instruction sets.

**Modern Compiler Implementation in C**
*Andrew Appel*

The implementation for closures came from this book. The ideas for Types as tree with `merging` and `unification` came from this book. Really good section on garbage collection.

**Engineering a Compiler**
*Keith D. Cooper*

The ideas for Struct Layout and multi-dimensional array access came from here.

**Compiler Principles, Techniques and Tools (The Dragon Book)**
*Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman*

Surprisingly very good section on garbage collection. Basic blocks were described in this book and this is the mechanism how try/catch handlers are type checked.

**The JVM Instruction Set**

This was the only reference I used to create my own instruction set.

# Computer Architecture Books

**Computer Architecture: A Quantitative Approach, Sixth Edition**
*John L. Hennessy (Stanford University), David A. Patterson*

A huge body of work on real world computer architecture. Thread level parallelism is really fun read that helps explain hardware problems with cache consistency.

**Memory Barriers: a Hardware View for Software Hackers**
*Paul E. McKenney*

An excellent paper describing memory barriers and fences. It will help you understand these materials well.

# Garbage Collection Books and Papers

**The Garbage Collection Handbook**
*Richard Jones, Antony Hosking, Eliot Moss*

Absolutely incredible publication going over 50 years worth of research with a great emphasis on parallel garbage collection. Everything I know about garbage collection is from this book. The book also has an excellent concurrency primer.

**Uniprocessor Garbage Collection Techniques**
*Paul R. Wilson*

A really gentle read into the main ideas in Garbage collection, complements really well with the garbage collection handbook.

**Dynamic Storage Allocation: A Survey and Critical Review**
*Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles*

An older paper but surprisingly this is the strongest material I found that explains allocation techniques. I just used the FreeList implementation.

**The Hoard Allocator**
*Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, Paul R. Wilson*

A paper on how to create a multi-threaded allocation. The ideas for a global page cache came from reading this paper.

**Doug Lea's Malloc**
https://gee.cs.oswego.edu/dl/html/malloc.html

A much more sophisticated allocator than mine using a combination of segregated lists and trees created by Doug Lee himself. If I am to improve the noC allocator I will attempt to implement Doug Lea's one because it seems simple.

**Windows Internals, Part 1**
*Pavel Yosifovich, Alex Ionescu, Mark Russinovich, David Solomon*

Heap Manager: A good section describing how the kernel provides a heap and how it internally works. The ideas for detecting heap corruption came from this book.

**Computer Systems: A Programmer's Perspective**
*Randal E. Bryant, David R. O'Hallaron*

If there was a one book to read it would be this one. Its section on Virtual Memory and allocation is excellent, actually all chapters are excellent.

**Powerful Page Mapping Techniques**
*Casey Muratori*
https://www.youtube.com/watch?v=H8THRznXxpQ

Excellent video showing you the power of virtual memory. I might even use what he has taught to try to implement a rewind debugger. Casey also has a performance aware programming course where he teaches how to utilise modern hardware.

## Scheduler Books and Publications

**Operating Systems: Three Easy Pieces**
*Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau*

A nice introduction to multi-level feedback queues and also one of my favourite books.

**Operating System Concepts**
*Abraham Silberschatz, Peter B. Galvin, Greg Gagne*

A nice introduction to different OS schedulers and also a great explanation for M:N schedulers, Scheduler Activation, Green Threads. Also has an excellent chapter on concurrency with great explanations into how monitors are implemented in Java.

---

**Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism**
*Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, Henry M. Levy*

I do not believe any OS supports scheduler activations anymore but a really fun paper to read about how a different kernel design supports a kind of non-blocking I/O. I do not think I fully understand this paper but was still a fun read.

---

**Windows Internals, 2nd Edition**
*Mark Russinovich, David Solomon*

Covers the Windows scheduler and explains its priority levels as well as how it boosts I/O. Also introduces some implementation details on vectored I/O.

---

**Parallelizing the Naughty Dog Engine Using Fibers**
*Christian Gyrling*
https://www.youtube.com/watch?v=HIVBhKj7gQU

Surprisingly simple how they managed to parallelize their engine. The PS4 I believe had 6 cores, this has led me to think at what core count does a simple scheduler fail to scale.

---

**Scalability! But at what COST?**
*Frank McSherry, Michael Isard, Derek G. Murray*

A great paper to read that introduces a metric called "COST": "What hardware configuration required before the platform outperforms a competent single-threaded implementation?" Now that the noC sandbox is ready (for the most part), I would like to see at what point the simple implementations break down.

---

# Concurrency and Locking Books

---

**The Art of Multiprocessor Programming**
*Maurice Herlihy, Nir Shavit*

The authoritative text on parallel algorithms. I did not end up using a lot of the CAS based algorithms in this book but nevertheless a phenomenal publication, and I might implement some of the non-blocking algorithms to improve the queue.

**Shared-Memory Synchronization**
*Michael L. Scott*

Chapter on Synchronization and Scheduling gave me the ideas for awaitable and futures. He also has two great videos on concurrency data structures; I have only seen part 1, the material in part 2 was too confusing for me.

- Michael L. Scott: Non-blocking Data Structures, Part 1
  https://www.youtube.com/watch?v=9XAx279s7gs
- Michael L. Scott: Non-blocking Data Structures, Part 2
  https://www.youtube.com/watch?v=cQIktrroRL0

**FreeBSD Internals**
*An Overview of Locking in the FreeBSD Kernel*
https://www.youtube.com/watch?v=UfiTV9QWhM4

**ANA L UCIA DE MOURA and ROBERTO IERUSALIMSCH**
*Revisiting Coroutines*

The paper that influenced me to support closures and would be the mechanism to support async co(no) routines.

**Introduction to mutli threading**
*Casey Muratori*
This is a playlist for the Handmade hero series where casey explains multi threading concepts.

https://www.youtube.com/watch?v=qkugPXGeX58&
list=PLEMXAbCVnmY7me6j4VtpCYMuZX3QpcBBH

# Incredible Compiler projects

**Jonathon Blow's Jai programming language**
*Jonathon Blow*

Jonathon Blow started work on his own programming langauge 10 years ago and is developing his new game on this language. His first two talks about ideas for a new programming langauge are fantastic

- https://www.youtube.com/watch?v=TH9VCN6UkyQ
- https://www.youtube.com/watch?v=5Nc68IdNKdg

---

**archimedes - C++ reflection via code generation**
*JDH*

JDH has an amazing project on youtube where he connects into the clang compiler to create a serialization library for C++

- https://www.youtube.com/watch?v=aJt2POa9oCM

---