ETCH

Define Once, Etch Forever.

A language that *keeps you safe* before it even runs,
yet feels like *you're just scripting* - **until it blows your mind!**

Why Etch? 🤔

The Origin Story Nobody Asked For (But You're Getting Anyway)

The Endless Quest: 🔍

- Spent *years* hunting for the "perfect" scripting engine.
- Lua? *Too bare.* Python? *Too slow.* JavaScript? *Too... JavaScript.*
- Tried them all. They all lacked *something*.
- Performance ⚡ vs Safety 🛡️ vs Simplicity 📝. You can only **pick two**.

Why Etch? 🤔

The Origin Story Nobody Asked For (But You're Getting Anyway)

The "Why Not?" Moment: 💡

- **Plot twist:** Compilers are actually **FUN** to build! 🎉
- Living in the age of AI: *"Make an app!"* 🤖
- Me: *"...What if I make a LANGUAGE instead?"* 🎯
- **YOLO Compiler Theory!**

Why Etch? 🤔

The Origin Story Nobody Asked For (But You're Getting Anyway)

The Mad Experiment: 🚀

- Built Etch as a *playground* for compiler tech
- Prove correctness? *Why not!*
- Safety without complexity? *Challenge accepted!*
- **Etch: Because life's too short** 🔥

The Bugs That Haunt Your Dreams

Those sneaky runtime gremlins - the ones that compile just fine, only to explode right when you hit play.

Numerical Safety Issues:

- ❌ Division by zero crashes
- ❌ Integer overflow vulnerabilities

Memory Safety Issues:

- ❌ Nullptr dereferencing
- ❌ Uninitialized variable bugs

Array and Bounds Issues:

- ❌ Array bounds errors
- ❌ Buffer overflows
- ❌ Platform-dependent behavior

Compile-Time Safety Verification

Etch proves safety properties at compile-time through static analysis.

Safety Guarantees:

- ✅ No division by zero
- ✅ No integer overflow
- ✅ No nullptr dereferences
- ✅ No uninitialized variables
- ✅ No array out of bounds errors

Additional Benefits:

- ✅ Dead code automatically eliminated
- ✅ Redundant checks removed
- ✅ Errors caught before deployment

👋 nuqneH, Etch!

```
fn main() -> void {
    print("Hello, World!");
}
```

C-like syntax you already know-*simple, clean, familiar*.

No need to learn *Klingon* to say hello! 🖖

🏗️ Language Architecture

Multi-stage compilation pipeline:

```
📄 Source Code
    ↓
🔍 Frontend (Lexer → Parser → AST)
    ↓
🎨 Type Checker (Inference + Validation)
    ↓
🛡️ Safety Prover (Range Analysis + Symbolic Execution)
    ↓
⚡ Comptime Executor (AST Transformation)
              ↓
    ┌─────────┴──────────┐
    ↓                    ↓
💾 Bytecode VM      🔧 C Code Generator
```

✅ **Each stage adds safety guarantees**

📋 The Compilation Pipeline

Staged transformation with safety guarantees at each step:

Stage	Process	Output
1️⃣	**Lexer**	Token stream (frontend/lexer)
2️⃣	**Parser**	AST (frontend/ast - frontend/parser)
3️⃣	**Type Checker**	Typed + Inferred AST (typechecker/)
4️⃣	**Safety Prover**	Proven AST (prover/)
5️⃣	**Comptime Executor**	Transformed AST (comptime/)
6️⃣	**Bytecode Generator**	Register VM bytecode (interpreter/)
7️⃣	**Cache**	Disk (`__etch__`/ directory)
8️⃣	**VM / C Backend**	Execution or native binary (backend/)

🔍 Frontend: Lexer → Parser

Token-based lexical analysis:

- 📍 Position tracking for error reporting
- ⚡ Efficient single-pass scanning
- 🎯 Source location preservation

Recursive descent parser:

- 🌳 Produces strongly-typed AST
- 📝 Expression kinds: Binary ops, Calls, Arrays, Pattern matching
- 🔀 Statement kinds: Declarations, Control flow, Assignments
- 🛡️ Syntax validation during parsing

🌳 AST Node Types

Expression variants (20+ types):

- 🔢 **Literals**: `ekInt`, `ekFloat`, `ekString`, `ekBool`, `ekChar`
- ➕ **Operations**: `ekBin`, `ekUn`, `ekCall`
- 📦 **Collections**: `ekArray`, `ekIndex`, `ekSlice`, `ekArrayLen`
- 🎭 **Pattern matching**: `ekMatch`, `ekOptionSome`, `ekResultOk`
- 🏗️ **Objects**: `ekObjectLiteral`, `ekFieldAccess`, `ekNew`

Statement variants (15+ types):

- 📌 **Declarations**: `skLet`, `skVar`, `skFun`, `skType`
- 🔀 **Control flow**: `skIf`, `skWhile`, `skFor`, `skReturn`
- ⚙️ **Others**: `skAssign`, `skDefer`, `skComptime`

🎯 Type System: Three-Phase Process

Phase 1: Type Collection 📚

- Gather all type definitions and global variables
- Build forward reference table
- Create type environment

🎯 Type System: Three-Phase Process

Phase 2: Type Inference 🔮

- Hindley-Milner style inference for generics
- Constraint generation and unification
- Return type inference from body
- Generic function instantiation on demand

🎯 Type System: Three-Phase Process

Phase 3: Type Validation ✅

- Expression type checking (expressions.nim)
- Statement type checking (statements.nim)
- Full program validation

🎨 Type Kinds

Primitive Types:

- ✅ `tkBool` (1 byte)
- 📝 `tkChar` (1 byte)
- 🔢 `tkInt` (8 bytes, signed)
- 🔢 `tkFloat` (8 bytes, double)
- 📝 `tkString`
- 🚫 `tkVoid`

Composite Types:

- 📦 `tkArray` - Fixed and dynamic arrays
- 🔗 `tkRef` - References with generational tracking
- 🏗️ `tkObject` - Structured data with fields

🎨 Type Kinds

Advanced Types:

- ❓ `tkOption` - Optional values (some/none)
- ✨ `tkResult` - Result types (ok/error)
- 🎭 `tkGeneric` - Generic type parameters
- 🔗 `tkUnion` - Sum types

🛡️ Prover: Safety Through Analysis

Safety analysis through symbolic execution

Core Components:

- 🔍 **expression_analysis**: Range propagation, expression evaluation
- ➕ **binary_operations**: Arithmetic operation range inference
- 🔀 **symbolic_execution**: Control flow analysis
- ⚡ **function_evaluation**: Pure function compile-time execution

Key Concept is tracking values

- known/unknown
- min/max range
- non-zero/non-null
- initialized/used/last use
- is array/string
- array/string size

Every variable has a proven state at every program point!

🔬 Prover: Analysis Phases

Phase 1: Environment Setup 📋

- Initialize environment with variable declarations
- Add all globals to environment

🔬 Prover: Analysis Phases

Phase 2: Global Analysis 🌍

- Analyze global variable initializations
- Track global value ranges

🔬 Prover: Analysis Phases

Phase 3: Function Analysis 🎯

- Prove main function first
- Analyze all reachable functions
- Validate safety at every operation

🔬 Prover: Analysis Phases

Phase 4: Property Validation ✅

- Check non-zero divisors (division/modulo)
- Verify array bounds safety
- Prevent integer overflow
- Ensure variable initialization

The Safety Prover in Action

```
// ✅ Safe!
fn main() -> void {
    let divisor: int = rand(5, 10);          // Range: [5, 10]
    let calculation: int = 100 / divisor;    // ✅ Safe!
    print(calculation);
}
```

Safe Example: divisor ranges [5, 10] → division succeeds ✅

The Safety Prover in Action

```
// ❌ COMPILE ERROR
fn main() -> void {
    let divisor: int = rand(5);            // Range: [0, 5]
    let calculation: int = 100 / divisor;   // ❌ COMPILE ERROR
    print(calculation);
}

/*
Compiling: xyz.etch
xyz.etch:4:32: error: cannot prove divisor is non-zero in main
3 |     let divisor: int = rand(5);            // Range: [0, 5]
4 |     let calculation: int = 100 / divisor;   // ❌ COMPILE ERROR
                                   ^
5 |     print(calculation);
*/
```

Unsafe Example: divisor ranges [0, 5] → *COMPILE ERROR* ❌

Compiler tracks value ranges and proves divisor should be non-zero.

📦 Arrays with Safety Guarantees

```
fn main() -> void {
    let numbers: array[int] = [10, 20, 30, 40, 50];

    let count: int = #numbers;               // Length operator
    let middle: int = numbers[count / 2];    // Bounds checked
    let slice = numbers[1:4];                // Safe slicing, inferred as array[int]
}
```

- ✅ Compile-time bounds checking when possible
- 🔍 Compiler enforces insertion of runtime checks when necessary
- 📝 Clear error messages

🎨 Type System & Inference

```
fn main() -> void {
    let x: int = 42;                // Explicit type
    let y = 3.14;                   // Inferred as float
    let name = "Etch";              // Inferred as string
    let numbers = [1, 2, 3];        // Inferred as array[int]
}
```

- ✨ Strong static typing
- 🔮 Smart type inference
- ✅ No surprises

🏗️ Types and Objects

Type Aliases, Unions and Object Definitions:

```
// Type aliases for clarity
type UserId = int;
type Email = distinct string;

// Union types for sum types
type IntOrString = int | string;

// Object types with fields
type Point = object {
  x: int;
  y: int;
};

fn main() -> void {
  let p: Point = { x: 10, y: 20 };
  print(p.x + p.y);  // 30
}
```

Safety Guarantees:

- 🛡️ All object fields must be initialized before use
- ✅ Prover tracks initialization of each field
- 🔍 Compile-time error if accessing uninitialized field

🎯 Uniform Function Call Syntax

Call functions as methods using dot notation:

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

fn main() {
  var x: int = 10;
  var y: int = 20;

  // Traditional call
  discard add(x, y);

  // UFCS — first argument becomes receiver
  var result2: int = x.add(y);
  print(result2); // 30

  // But also!
  print(5.add(15)); // 20
}
```

Clean, readable method-style calls without OOP overhead! 🚀

🎨 Pattern Matching: Option & Result

Safe value extraction with exhaustive matching:

```
fn divideInts(a: int, b: int) -> result[int] {
  if b == 0 {
    return error("Division by zero");
  } else {
    return ok(a / b);
  }
}

fn main() -> void {
  let divResult: result[int] = divideInts(42, 6);
  let message: string = match divResult {
    ok(value) => {
      "Success: " + toString(value);
    }
    error(err) => {
      "Failed: " + err;
    }
  };
  print(message);  // "Success: 7"
}
```

🎨 Pattern Matching: Advanced

Option types and nested patterns:

```
fn tryGetElement(arr: array[int], index: int) -> option[int] {
  if index >= 0 and index < #arr {
    return some(arr[index]);
  } else {
    return none;
  }
}

fn main() -> void {
  let numbers: array[int] = [10, 20, 30];
  let maybeValue: option[int] = tryGetElement(numbers, 1);

  let result: string = match maybeValue {
    some(value) => "Found: " + toString(value);
    none => "Not found";
  };
  print(result);  // "Found: 20"
}
```

Compiler enforces exhaustive pattern coverage! ✅

⏰ Defer: Guaranteed Cleanup

Execute cleanup code when scope exits:

```
fn main() {
  print("Start");

  defer { print("Cleanup - runs last!"); }

  print("Middle");

  defer { print("Second cleanup"); }

  print("End");
}
```

Output:

```
Start
Middle
End
Second cleanup
Cleanup - runs last!
```

Defers execute in reverse order (LIFO) - Perfect for resource cleanup! 🧹

📦 Import System: Modules & CFFI

Module Imports (Etch code):

```
import math
import math { sqrt, pow }
```

C FFI Imports (Native libraries):

```
import ffi cmath {
  fn sin(x: float) -> float;
  fn cos(x: float) -> float;
  fn sqrt(x: float) -> float;
}

fn main() -> void {
  var pi: float = 3.14159;
  var sine: float = sin(pi / 2.0);
  print(sine);  // ~1.0
}
```

Zero-cost abstractions:

- 🚀 Direct C function calls (no overhead)
- 🔗 Dynamic library loading (runtime)
- ✅ Type-safe FFI boundaries

⚡ Compile-Time Execution

Comptime evaluation during compilation:

Not macros or templates-actual code execution in the compiler.

Comptime Use Cases

- 🎯 **Build-time configuration**: Different builds from same source
- 📊 **Lookup tables**: Compute once at compile-time, use at runtime
- 🚩 **Feature flags**: Conditional compilation based on environment
- 📦 **Resource embedding**: Templates, shaders, assets in binary
- 🏷️ **Version information**: Embed git commit hash, build date
- 🌐 **Platform-specific code**: Single codebase for multiple targets

Comptime Basics

```
fn square(x: int) -> int {
    return x * x;
}

fn main() -> void {
    // Prints 64 during compilation
    comptime{ print(square(8)); }

    // Evaluates to constant at compile-time
    let sq: int = comptime(square(8));
    print(sq);
}
```

Zero runtime overhead

- Function calls happen during compilation
- Results embedded as constants in bytecode
- No function call overhead at runtime

Comptime Blocks

```
fn main() -> void {
    comptime {
        print("Hello from the compiler!");

        var i: int = 0;
        while i < 5 {
            i = i + 1;
        }

        print(i);
    }

    print("Hello from runtime!");
}
```

Compile-time output:

```
Hello from the compiler!
5
```

Runtime output:

```
Hello from runtime!
```

File Embedding

```
fn main() -> void {
    // File read at COMPILE-TIME
    let config: string = comptime(readFile("config.txt"));
    print(config);  // File embedded in binary!
}
```

Embed files directly into your binary

- No runtime I/O
- No missing file errors
- Single executable deployment

Code Injection

```
fn main() -> void {
    comptime {
        let env: string = readFile(".env");

        if env == "production" {
            inject("LOG_LEVEL", "int", 0);
            inject("DEBUG", "bool", false);
        } else {
            inject("LOG_LEVEL", "int", 2);
            inject("DEBUG", "bool", true);
        }
    }

    if DEBUG { // Variable was injected!
        print("Debug mode enabled");
    }
}
```

Metaprogramming without macros, generate code based on compile-time conditions

🔀 Two Execution Modes

Single source, dual execution paths:

```
Etch Compiler
     ↓
Bytecode ───────→ C Backend
     ↓                 ↓
Bytecode VM       C Compiler
     ↓                 ↓
Execution         Execution
```

🔧 **Bytecode VM:** Fast iteration, debugger support, portable caching

🚀 **C Backend:** Native performance, compiler optimizations, standalone binary

Development workflow: 🧪 VM for rapid → 📦 C for deployment

🔧 Bytecode VM - Fast Development

Perfect for development iteration:

- ⚡ Instant execution (cached)
- 🐛 Full debugger support
- 📦 Portable bytecode
- 🚫 No C compiler needed
- 🔗 FFI to C libraries (runtime)
- 📝 Rich runtime errors

🚀 C Backend - Maximum Performance

Compiles to clean, readable C code:

- 🎯 Native machine code via gcc/clang
- ⚡ Platform optimizations (-O3)
- 🔧 System calling conventions
- 🔗 FFI to C libraries (linktime)
- 📦 Standalone executable

🎛️ Register-Based VM Architecture

⚡ RegVM: Lua-inspired register machine ⚡

Stack VM	Register VM
PUSH 5	LoadK r0, 5
PUSH 3	LoadK r1, 3
ADD	Add r2, r0, r1
POP r0	

Architecture Details:

- 📟 256 registers per function frame (8-bit addressing)
- 💾 65536 constants per function (16-bit index)
- 🎯 3-address instruction format (A = B op C)
- 🔧 Multiple instruction encodings: ABC, ABx, AsBx, Ax
- ⚡ Fused instructions for common patterns

💾 Bytecode Instructions

Instruction Categories:

Category	Instructions
📦 Load/Store	LoadK, Move, GetGlobal, SetGlobal
🧮 Arithmetic	Add, Sub, Mul, Div, Mod, Neg
🔢 Comparison	Eq, Lt, Le, Gt, Ge, Ne
🎯 Control Flow	Jump, JumpIf, JumpIfNot, TestJump
📞 Function Calls	Call, Return
⚡ Fused Instructions	AddAdd, MulAdd, LoadAddStore, EqStore

💾 Bytecode Caching

Performance improvement: 🚀 faster subsequent runs

First run:

`Source → Parse → Typecheck → Prove → Compile → Cache`

Subsequent runs:

`Source Hash Check → Load Cached Bytecode → Run`

Cache invalidation:

- ✅ Source file changed → recompile
- ✅ Source hash mismatch → recompile
- ✅ Bytecode version changed → recompile
- ✅ Compiler binary changed → recompile
- ✅ Compiler flags changed → recompile

👑 Etch compiler is written in Nim 🎯

Why Nim for compiler development:

- ✨ Compiles to C → Portable, fast execution
- ✨ Python-like syntax → Readable codebase
- ✨ Zero-cost abstractions → Efficient compilation
- ✨ Memory safe with no GC → No crashes and leaks
- ✨ Strong standard library → Less boilerplate
- ✨ Metaprogramming → DSL capabilities

Benefits for Etch:

- 🌳 Clean AST representation with algebraic types
- 🔄 Pattern matching for compiler passes
- 🔗 Easy C FFI for library integration
- ⏱️ Fast compilation of the compiler itself (< 6 seconds)

🐛 VSCode Debugger Integration

Full DAP (Debug Adapter Protocol) Support ✨

Features:

- 🔴 Set breakpoints in .etch files
- ⏯️ Step through execution (step in/out/over)
- 📊 View call stack
- 🔍 Inspect variables
- 👁️ Watch expressions (in progress)
- 📌 Conditional breakpoints (in progress)

Debug Server:

- 🔌 DAP protocol implementation (console based)
- 🌐 TCP/IP communication (in progress)
- 🎛️ Integrated with RegVM

📊 Benchmark Results

Real benchmark data from `hyperfine` on an M3 (generated 2025-10-22)

📊 Benchmark Results

Real benchmark data from `hyperfine` on an M3 (generated 2025-10-22)

Benchmark	C	VM	Python 3	C vs Py	VM vs Py
🧮 Arithmetic ops	6.5ms	115.8ms	103.5ms	**15.9x**	0.9x
📦 Array ops	6.9ms	32.9ms	42.8ms	**6.2x**	**1.3x**
🔁 For loops	10.7ms	18.1ms	39.4ms	**3.7x**	**2.2x**
📞 Function calls	14.2ms	56.8ms	32.9ms	**2.3x**	0.6x
🔢 Math intensive	5.0ms	26.8ms	31.2ms	**6.2x**	**1.2x**
💾 Memory alloc	2.6ms	11.3ms	23.6ms	**9.0x**	**2.1x**
🔄 Nested loops	6.6ms	66.0ms	40.3ms	**6.1x**	0.6x
📝 String ops	13.9ms	10.3ms	25.2ms	**1.8x**	**2.4x**

Key Takeaway: C backend ~1.8-15.9x faster than Python, VM competitive for many workloads

🚀 More Optimization Opportunities

Current Optimizations:

- ✅ Constant folding (partial)
- ✅ Dead code elimination
- ✅ Range-based check elimination
- ✅ Fused instructions
- ✅ Bytecode caching

Future Optimizations:

- 🔄 Loop hoisting optimizations
- 🧮 Common subexpression elimination
- 🎨 Type-specialized instructions
- 📦 Function inlining
- 🎯 Register coalescing

The compiler keeps getting faster!

🧪 Development Experience

Etch provides multiple tools for exploration:

Experimentation:

- ⚡ Test comptime evaluation limits
- 🔬 Understand prover range analysis
- 📊 Profile VM vs C backend performance
- 🎨 Generate code through metaprogramming

Tooling:

- 🎨 VSCode extension with syntax highlighting
- 🐛 DAP debugger integration
- 📝 Verbose logging for compiler internals
- 📈 Performance benchmarking with `just perf`

Learning compiler technology:

- 🔍 Inspect bytecode with verbose mode
- 🛡️ Study prover analysis output
- 📄 Compare generated C code
- 🚀 Understand optimization passes

🎯 Who Is Etch For?

Perfect for:

- 🔬 Compiler enthusiasts exploring PL design
- 🛡️ Programmers trying safety without complexity
- 🎮 Game developers needing a fun scripting runtime
- 📚 Discovering program verification

🎯 Current Status

Language Status: Active Development 🚧

What works:

- ✅ Core language features
- ✅ Safety prover with range analysis
- ✅ Compile-time execution
- ✅ Bytecode VM with caching
- ✅ C code generation backend
- ✅ VSCode debugger integration
- ✅ Test framework
- ✅ Performance benchmarking

Production ready? Not yet! ⚠️

Great for: 🧪 Experiments · 📚 Learning · 🔬 Research

🗺️ Optimization Roadmap

Phase 1: Bytecode optimization 💾

- 🔧 Re-enable optimizer
- 📊 Enhanced constant folding
- 🛡️ Integrate prover data into compiler

Phase 2: Instruction improvements 🎯

- 🏷️ Jump target tables
- 📝 ARG instructions
- 🔄 Reversed operations

🗺️ Roadmap

Phase 3: Advanced optimizations 🚀

- 🔬 Peephole optimization
- 🧮 Common subexpression elimination
- 🔄 Loop optimizations

Phase 4: Type-aware optimization 🎨

- 🎯 Static type specialization
- 📦 Function inlining

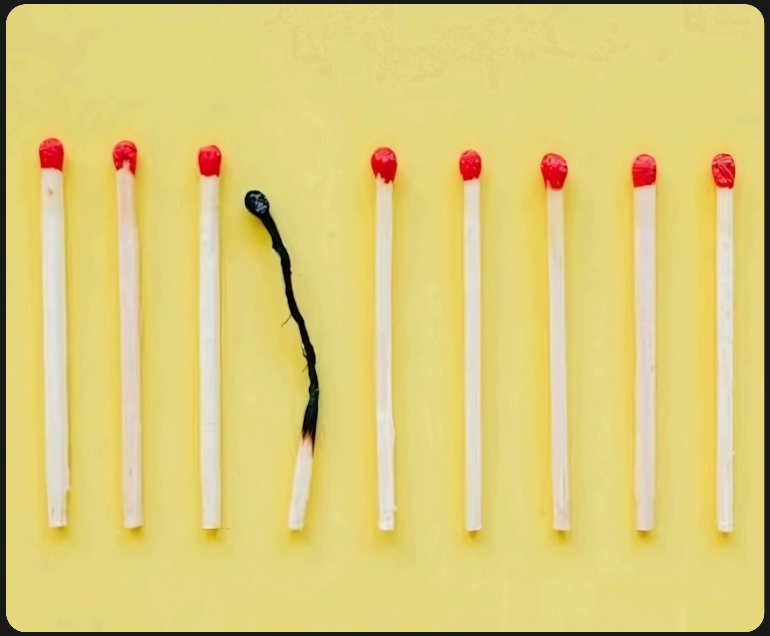
Let's see Etch in action! 🎬

Demos:

1. 🛡️ Safety proofs catching bugs
2. ⚡ Comptime execution
3. 🐛 Debugger in VSCode
4. 📊 Performance comparison
5. 🔧 C backend code generation

Questions?

"Define once, Etch forever."



Thank you! 🚀