Compiler - A Backend

Jarrod Connolly

April 3, 2026

Abstract

Complect grew from a Babel-based transpiler into a multi-backend compiler with an LLVM code generator, user-defined functions, manual memory management, and SDL graphics. This article is the architecture story behind that transformation. I cover how the backends split, why LLVM entered the picture, and what it took to go from emitting JavaScript to generating something that can draw a rotating cube.

Introduction

Back in 2024, I wrote about building a toy compiler in Node.js. That article covered the lexer, the parser, and the Babel AST, the front half of a compiler, stitched together with Node.js streams. It was a solid foundation, but it was also incomplete. A transpiler that only speaks JavaScript is more of a translator than a compiler. I wanted Complect to generate native code.

Over a concentrated stretch in late 2025, I rebuilt the core architecture, added an LLVM backend, introduced functions and manual memory management, and wired up SDL2 for graphics. The project went from roughly 1,200 lines of streams-based code to nearly 1,900 lines across four pipeline stages, two backends, and a VS Code extension. But the line count matters less than what changed underneath: the entire processing model, the intermediate representation, and the scope of what the language could express.

This article is about those architectural changes. I will walk through why I replaced streams with async generators, how the LLVM backend generates native binaries, what it took to add functions and memory management to the language, and how SDL graphics gave the project a visual payoff. There is more to cover (arrays, a Doom-style fire effect, and a proper language server) but those belong in a follow-up. For now, let us focus on the bones of the system.

You can follow along with the code at jarrodconnolly/complect.

From Streams to Async Generators

The original compiler used Node.js streams. The `pipeline` function from `node:stream/promises` connected the preprocessor to the tokenizer to the AST builder to an output buffer. Here is what that looked like:

```
import { pipeline } from 'node:stream/promises';

export async function compile(inputStream) {
  const preprocessor = new Preprocessor();
  const tokenizer = new Tokenizer();
```

```
  const babelAST = new BabelAST();
  const streamBuffer = new StreamBuffer();

  await pipeline(
    inputStream,
    preprocessor,
    tokenizer,
    babelAST,
    streamBuffer,
  );

  return {
    code: streamBuffer.buffer,
    preprocessorTokenCount: preprocessor.tokenCount,
    tokenCount: tokenizer.tokenCount,
    astNodeCount: babelAST.astNodeCount,
  };
}
```

Each stage extended `Transform` from the `node:stream` module, overriding `_transform(chunk, encoding, callback)` to push tokens downstream. It worked. The modularity was real: I could insert new stages between existing ones, test each in isolation, and reason about data flow as a pipeline.

But there was friction. Every stage had to manage its own internal buffer of incomplete data. The preprocessor, for example, reads characters one at a time. A multi-character operator like `==` spans two characters, so the transform had to hold state across chunks. The tokenizer had similar issues with multi-character tokens. Streams also pushed error handling into callbacks and made debugging awkward: stepping through a `_transform` method attached to a pipeline gives you a lot of Node.js internals before you reach your own code.

Async generators solved all of this. Instead of extending `Transform`, each stage is now a simple async generator function:

```
async *process(preprocessorGen) {
  for await (const preprocessingToken of preprocessorGen) {
    // classify, validate, yield tokens
    yield this._createToken(tokenType, value, line, column);
  }
}
```

The compiler entry point changed from a pipeline to a sequence of `for await...of` iterations:

```
export async function compile(inputStream, backend = 'babel') {
  const preprocessor = new Preprocessor();
  const tokenizer = new Tokenizer();
  const astBuilder = new ASTBuilder();

  const preprocessorGen = preprocessor.process(inputStream);
  const tokenizerGen = tokenizer.process(preprocessorGen);
  const ir = await astBuilder.build(tokenizerGen);

  // dispatch to backend...
}
```

Each generator pulls from the previous one, yielding results as they become available. The control flow is explicit. You can `console.log` between yields. You can wrap a single stage in a try/catch without touching the others. And because generators are lazy, you still get the memory efficiency of streams: only the current token or AST node is in memory at any time, not the entire file.

This was not a performance decision. Streams are faster for truly large data. But Complect compiles toy programs, a few hundred lines at most, and the clarity gain from async generators far outweighs any microsecond difference in throughput. For a learning project, being able to read the data flow from top to bottom in a single file is worth more than pipeline abstraction.

The trade-off is that async generators are not drop-in replacements for streams. You lose backpressure handling and the ability to pipe directly to file descriptors. But for Complect, the compiler reads from a file or stdin, processes everything in memory as a batch of tokens, and writes to stdout or a file. Backpressure was never the bottleneck. Understanding the code was.

The LLVM Backend

The original Complect had one output: JavaScript. The Babel translator walked the intermediate representation and produced a Babel AST, which Babel's `@babel/generator` then printed as JavaScript source. This was a pragmatic starting point. Babel handles the messy parts of code generation (operator precedence, statement formatting, scope management) and I could focus on getting the parser right.

But generating JavaScript from a custom language is only half the story. I wanted Complect programs to compile to native binaries. That meant generating something a real compiler toolchain could consume: LLVM Intermediate Representation.

The Pluggable Backend Design

The architecture now has a clean seam at the IR layer. The AST builder produces an intermediate representation made up of classes like `VariableDeclaration`, `BinaryExpression`, `IfStatement`, and `WhileStatement`. These nodes are language-agnostic: they describe what the program does without committing to how it will be executed.

After the IR is built, `compiler.js` dispatches to one of two translators:

```
if (backend === 'llvm') {
  const llvmTranslator = new LLVMTranslator();
  result = { code: llvmTranslator.translate(ir) };
} else {
  const babelTranslator = new BabelTranslator();
  result = babelTranslator.translate(ir);
}
```

Both translators accept the same IR. The Babel translator maps IR nodes to Babel AST nodes (`VariableDeclaration` becomes `t.variableDeclaration('let', ...)`). The LLVM translator maps them to LLVM API calls via the `llvm-bindings` package. Adding a third backend (WebAssembly via Binaryen, or a custom interpreter) means writing one new translator class that accepts the same IR. The frontend does not change.

What LLVM IR Looks Like

The LLVM backend is the heavyweight. At 1,188 lines, it is the largest file in the project. It uses `llvm-bindings`, a Node.js native addon that wraps the LLVM C++ API, to build modules, functions, basic blocks, and instructions programmatically.

A simple Complect program like this:

```
make a 0
make b 1
make n 12
as n > 0
  n = n - 1
  print a
  make t a
  a = b
  b = b + t
repeat
```

Produces LLVM IR that looks like this:

```
define i32 @main() {
entry:
  %a = alloca i32, align 4
  store i32 0, i32* %a, align 4
  %b = alloca i32, align 4
  store i32 1, i32* %b, align 4
  %n = alloca i32, align 4
  store i32 32, i32* %n, align 4
  br label %cond

cond:
  %n1 = load i32, i32* %n, align 4
  %gt = icmp sgt i32 %n1, 0
  br i1 %gt, label %body, label %exit

body:
  ; ... loop body with arithmetic and print calls ...
  br label %cond

exit:
  ret i32 0
}
```

Every Complect variable becomes an `alloca` (a stack allocation). Arithmetic expressions become SSA instructions like `add`, `sub`, and `mul`. The `as`/`repeat` loop becomes a `cond` basic block that branches to either `body` or `exit`. The `print` statement becomes a call to C's `printf` with a format string like `"%d\n"`.

The LLVM backend declares runtime functions (`printf`, `malloc`, `free`, `strcpy`, `strcmp`, and others) as external declarations at the top of the module. When you compile the generated `.ll` file with `clang`, the linker resolves these against libc.

Compiling to a Binary

The workflow for native compilation is three steps:

```
# Generate LLVM IR from Complect source
complect --file fib.cplct --backend llvm --output fib.ll
```

```
# Compile IR to a native binary
clang fib.ll -o fib

# Run it
./fib
```

Seeing a program written in my own language produce real integers on stdout through a native binary (not through Node.js, not through a VM) was the moment the project felt like a real compiler.

Functions: A Language Grows Up

The original Complect language had no functions. Every program was a flat sequence of variable declarations, assignments, loops, and print statements. For a toy language powering FizzBuzz and Fibonacci, that was enough. But the limitations became obvious quickly: without functions, you cannot structure a program beyond trivial size.

Adding functions meant changes at every layer of the compiler.

Parsing

The parser already used an LL(1) approach: one token of lookahead, with each statement type identified by its first keyword. Functions fit this model cleanly. When the parser encounters `func`, it reads the function name, collects parameter identifiers, then parses the body as a block until it hits `end`:

```
if (token.value === 'func') {
  const nameToken = this.tokens[this.index++];
  const params = [];
  while (this.tokens[this.index].type === TokenType.identifier) {
    params.push(this.tokens[this.index++].value);
  }
  const body = this.parseBlock('end');
  return new FunctionDeclaration(nameToken.value, params, body, loc);
}
```

Function calls use the `call` keyword, with an optional `into` clause to capture the return value:

```
func double n
  make result 0
  result = n * 2
  return result
end

call double 21 into answer
print answer
```

The parser treats `call` as another keyword-triggered statement, reading the function name, collecting argument expressions, and optionally reading `into` followed by a result variable name.

LLVM Translation

Translating functions to LLVM IR is more involved than translating flat statements. Each `FunctionDeclaration` becomes a separate `llvm.Function` object with its own entry basic block, parameter variables, and a return instruction. The translator switches its insertion point to the new function, translates the body statements, then switches back to the calling context.

Function calls require knowing the parameter types at the call site. Since Complect is dynamically typed, the translator infers types by analyzing all call sites before generating code. The `collectFunctionSignatures` method walks the IR, records the types of arguments passed to each function, and builds a signature map. If a function is called with an integer argument in one place and a string in another, string wins (since strings are the superset case for the printing functions). This is not a type system, it is a heuristic, but it works for the current language.

The primes program demonstrates functions well:

```
func checkPrime n
  make i 2
  make result 1
  if n < 2
    result = 0
  endif
  as i * i <= n
    if n % i == 0
      result = 0
    endif
    i = i + 1
  repeat
  return result
end

make num 2
make limit 1024
as num <= limit
  call checkPrime num into isPrime
  if isPrime == 1
    print num
  endif
  num = num + 1
repeat
```

This compiles through the LLVM backend into a native binary that prints primes up to 1024. The `checkPrime` function becomes its own LLVM function, called once per iteration of the outer loop.

Memory Management

JavaScript developers rarely think about memory. Values are allocated, passed around, and eventually collected by the garbage collector. When targeting native code, that safety net disappears. Every allocation needs a corresponding deallocation, or the program leaks.

Complect has two features for memory management: explicit `free` and automatic string cleanup.

Explicit Free

The `free` statement deallocates a variable:

```
make name "Jarrod"
print name
free name
```

In the LLVM backend, `free` calls the C standard library `free()` function on the variable's stored pointer. It is simple, but it puts the responsibility on the programmer, exactly the kind of low-level control I wanted Complect to teach.

Automatic String Cleanup

Strings are trickier than integers. An integer fits in a register or a 32-bit stack slot. A string is a pointer to a heap-allocated buffer. When you reassign a string variable, the old buffer must be freed, or it leaks. The LLVM backend handles this automatically on assignment: before storing a new string value, it loads the old pointer and calls `free()` on it.

```
// In translateStatement, for AssignmentExpression with string type:
const oldValue = this.builder.CreateLoad(
  this.builder.getInt8PtrTy(), varInfo.value
);
this.builder.CreateCall(
  this.module.getFunction('free'), [oldValue]
);
this.builder.CreateStore(finalValue, varInfo.value);
```

This is not garbage collection. It is deterministic, immediate cleanup at the point of reassignment. It cannot handle cycles or shared references. But for a language without heap-allocated data structures beyond strings and arrays, it covers the common case.

The Babel backend takes a different approach: `free` compiles to setting the variable to `null`, relying on JavaScript's garbage collector to handle the rest. The same Complect program, compiled through different backends, has different memory semantics. That tension, between high-level convenience and low-level control, is one of the most interesting things about supporting multiple backends.

We Have Graphics: SDL and the Rotating Cube

The most viscerally satisfying addition to Complect was graphics. Using SDL2, programs compiled through the LLVM backend can open windows, draw pixels, and render 3D geometry.

How SDL Integration Works

The LLVM backend declares SDL2 functions as external C functions, the same way it declares `printf` and `malloc`:

```
const sdlInitType = llvm.FunctionType.get(
  this.builder.getInt32Ty(),
  [this.builder.getInt32Ty()],
  false
);
this.module.getOrInsertFunction('SDL_Init', sdlInitType);
```

When the translator encounters an `sdlInit` statement in the IR, it emits a call to `SDL_Init(SDL_INIT_VIDEO)`. When it encounters `sdlWindow`, it calls `SDL_CreateWindow` with the title, dimensions, and flags. Each SDL keyword in Complect maps one-to-one to an SDL2 C function call in the generated LLVM IR.

At link time, `clang` resolves these symbols against the system's SDL2 library:

```
clang sdl-cube.ll -lSDL2 -o sdl-cube
```

The Rotating Cube

The demo program is a wireframe cube that rotates in 3D space. It uses Complect's `sin` and `cos` built-in functions to compute perspective projection for each vertex:

```
func calcX x y z s c dist focal
  make rx 0
```

```
  rx = x * c - z * s
  rx = rx / 1000

  make rz 0
  rz = x * s + z * c
  rz = rz / 1000

  make temp 0
  temp = rz + dist

  make sx 0
  sx = 400 + rx * focal / temp

  return sx
end
```

The `sin` and `cos` functions take an angle in degrees and a scale factor, returning an integer. This fixed-point math avoids floating-point operations (Complect has no float type) while still producing smooth rotation. Each frame, the cube's eight vertices are projected to screen coordinates, and `sdlDrawLine` connects them to form the wireframe.

The full program is about 200 lines of Complect. It defines helper functions for projection, a `drawCube` function that takes a frame number and draws the rotated geometry, and a main loop that increments the frame, clears the screen, draws, and presents. The result is a smooth, real-time 3D animation compiled from a language I designed myself.

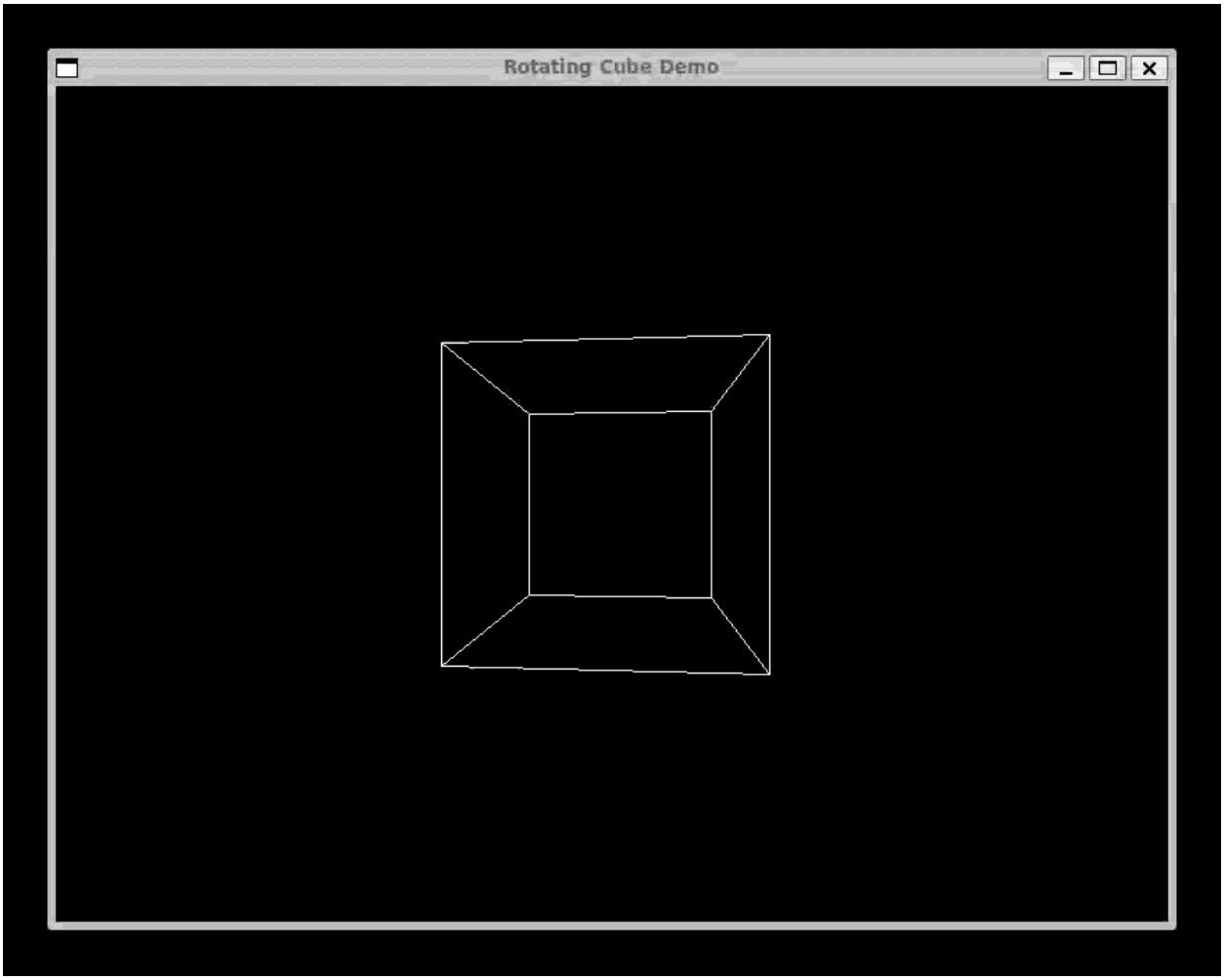

Figure 1: Rotating Cube Demo

This was the moment Complect stopped being a compiler exercise and started being a tool that produced something tangible, something you could see moving on screen. That shift in motivation is hard to overstate. When your compiler's output is a text file of LLVM IR, progress is abstract. When it is a spinning cube, progress is undeniable.

Lessons and What is Next

Rebuilding Complect's architecture taught me a few things that I did not fully appreciate when I started.

First, **simplicity in the processing model pays off**. Streams were the right choice for a talk about Node.js streams. Async generators are the right choice for a project you want to understand at a glance. The difference between the two `compile()` functions (one a pipeline of five stream constructors, the other a sequence of three `for await` loops) is the difference between code you explain and code that explains itself.

Second, **an intermediate representation is a force multiplier**. The IR layer decouples the frontend from the backends. Adding the LLVM backend did not require touching the lexer or parser. Adding functions required changes to the parser and both backends, but the IR structure kept those changes parallel and testable. If I add a WebAssembly backend next, I write one new translator class.

Third, **targeting native code forces you to confront things JavaScript hides**. Memory management, type representation, string handling, linking against C libraries: these are all invisible when your output is JavaScript. Generating LLVM IR made me think about what my language actually means at runtime, not just what it looks like in source form.

The project is not done. Since the changes described here, I have added array support, a frame buffer-based fire effect inspired by the classic Doom algorithm, and a proper LSP language server for VS Code. Those deserve their own article.

Complect remains a learning project, a place to explore how compilers work by building one, breaking it, and building it better. If you are curious about compilers but intimidated by the textbooks, I recommend starting small. Pick one backend. Parse one statement type. Generate one working program. The rest grows from there.